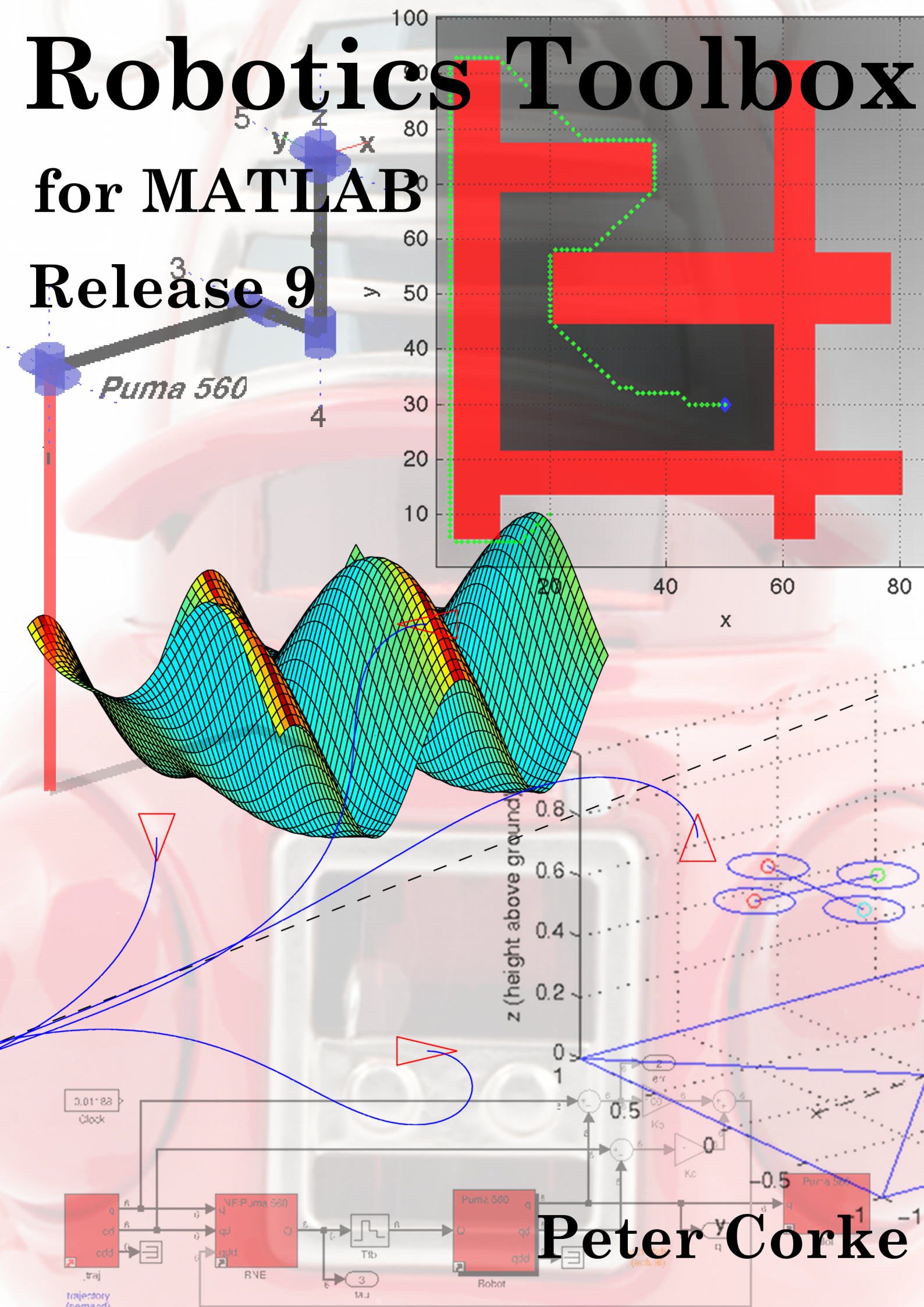
Robotics Toolbox

for MATLAB

Release 9

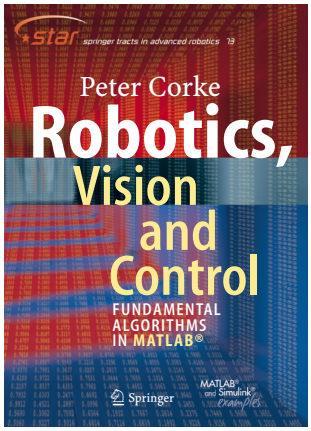


Peter Corke

Licence	LGPL
Toolbox home page	http://www.petercorke.com/robot
Discussion group	http://groups.google.com.au/group/robotics-tool-box

peter.i.corke@gmail.com
September 2011
http://www.petercorke.com

Preface



This, the ninth release of the Toolbox, represents over fifteen years of development and a substantial level of maturity. This version captures a large number of changes and extensions generated over the last two years which support my new book "*Robotics, Vision & Control*" shown to the left.

The Toolbox has always provided many functions that are useful for the study and simulation of classical arm-type robotics, for example such things as kinematics, dynamics, and trajectory generation. The Toolbox is based on a very general method of representing the kinematics and dynamics of serial-link manipulators. These parameters are encapsulated in MATLAB® objects — robot objects can be created by the user for any serial-link manipulator and a number of examples are provided for well know robots such as the Puma 560 and the Stanford arm amongst others. The Toolbox also provides functions for manipulating and converting between datatypes such as vectors, homogeneous transformations and unit-quaternions which are necessary to represent 3-dimensional position and orientation.

This ninth release of the Toolbox has been significantly extended to support mobile robots. For ground robots the Toolbox includes standard path planning algorithms (bug, distance transform, D*, PRM), kinodynamic planning (RRT), localization (EKF, particle filter), map building (EKF) and simultaneous localization and mapping (EKF), and a Simulink model a of non-holonomic vehicle. The Toolbox also including a detailed Simulink model for a quadcopter flying robot.

The routines are generally written in a straightforward manner which allows for easy understanding, perhaps at the expense of computational efficiency. If you feel strongly about computational efficiency then you can always rewrite the function to be more efficient, compile the M-file using the Matlab compiler, or create a MEX version.

The manual is now auto-generated from the comments in the MATLAB® code itself which reduces the effort in maintaining code and a separate manual as I used to — the downside is that there are no worked examples and figures in the manual. However the book "*Robotics, Vision & Control*" provides a detailed discussion (over 600 pages, nearly 400 figures and 1000 code examples) of how to use the Toolbox functions to

solve many types of problems in robotics, and I commend it to you.

Contents

Chapter 1

Introduction

1.1 What's new

Changes:

- The manual (robot.pdf) no longer contains a per function description. All documentation is now in the m-file, making maintenance and consistency easier.
- The Functions link from the Toolbox help browser lists all functions with hyperlinks to the indiviual help entries.
- The Robot class is now named SerialLink to be more specific.
- Almost all functions that operate on a SerialLink object are now methods rather than functions, for example plot() or fkine(). In practice this makes little difference to the user but operations can now be expressed as robot.plot(q) or plot(robot, q). Toolbox documentation now prefers the former convention which is more aligned with object-oriented practice.
- The parametrers to the Link object constructor are now in the order: theta, d, a, alpha. Why this order? It's the order in which the link transform is created: RZ(theta) TZ(d) TX(a) RX(alpha).
- All robot models now begin with the prefix mdl_, so puma560 is now mdl_puma560.
- The function drivebot is now the SerialLink method teach.
- The function ikine560 is now the SerialLink method ikine6s to indicate that it works for any 6-axis robot with a spherical wrist.
- The link class is now named Link to adhere to the convention that all classes begin with a capital letter.
- The quaternion class is now named Quaternion to adhere to the convention that all classes begin with a capital letter.
- A number of utility functions have been moved into the a directory common since they are not robot specific.

- skew no longer accepts a skew symmetric matrix as an argument and returns a 3-vector, this functionality is provided by the new function vex.
- tr2diff and diff2tr are now called tr2delta and delta2tr
- ctraj with a scalar argument now spaces the points according to a trapezoidal velocity profile (see lspb). To obtain even spacing provide a uniformly spaced vector as the third argument, eg. linspace(0, 1, N).

New features:

- Model of a mobile robot, Vehicle, that has the "bicycle" kinematic model (car-like). For given inputs it updates the robot state and returns noise corrupted odometry measurements. This can be used in conjunction with a "driver" class such as RandomPath which drives the vehicle between random waypoints within a specified rectangular region.
- Model of a laser scanner RangeBearingSensor, subclass of Sensor, that works in conjunction with a Map object to return range and bearing to invariant point features in the environment.
- Extended Kalman filter EKF can be used to perform localization by dead reckoning or map featuers, map buildings and simultaneous localization and mapping.
- Path planning classes: distance transform DXform, D* lattice planner Dstar, probabilistic roadmap planner PRM, and rapidly exploring random tree RRT.
- The RPY functions tr2rpy and rpy2tr assume that the roll, pitch, yaw rotations are about the X, Y, Z axes which is consistent with common conventions for vehicles (planes, ships, ground vehicles). For some applications (eg. cameras) it useful to consider the rotations about the Z, Y, Z axes, and this behaviour can be obtained by using the option 'zyx' with these functions (note this is the pre release 8 behaviour).
- jsingu
- jsingu
- lspb
- tpoly
- qplot
- mtraj
- mstraj
- wtrans
- se2
- se3
- trprint compact display of a transform in various formats.
- trplot
- trplot2 as above but for SE(2)
- tranimate

- DHFactor a simple means to generate the Denavit-Hartenberg kinematic model of a robot from a sequence of elementary transforms.
- Monte Carlo estimator ParticleFilter.
- vex performs the inverse function to skew, it converts a skew-symmetric matrix to a 3-vector.
- Pgraph represents a non-directed embedded graph, supports plotting and minimum cost path finding.
- Polygon a generic 2D polygon class that supports plotting, intersectio/union/difference of polygons, line/polygon intersection, point/polygon containment.
- plot_box plot a box given TL/BR corners or center+WH, with options for edge color, fill color and transparency.
- plot_circle plot one or more circles, with options for edge color, fill color and transparency.
- plot_sphere plot a sphere, with options for edge color, fill color and transparency.
- plot_ellipse plot an ellipse, with options for edge color, fill color and transparency.
- plot_ellipsoid plot an ellipsoid, with options for edge color, fill color and transparency.
- plot_poly plot a polygon, with options for edge color, fill color and transparency.
- about one line summary of a matrix or class, compact version of whos
- tb_optparse general argument handler and options parser, used internally in many functions.

Bugfixes:

- Improved error messages in many functions
- Removed trailing commas from if and for statements

1.2 Support

There is no support! This software is made freely available in the hope that you find it useful in solving whatever problems you have to hand. I am happy to correspond with people who have found genuine bugs or deficiencies but my response time can be long and I can't guarantee that I respond to your email. I am very happy to accept contributions for inclusion in future versions of the toolbox, and you will be suitably acknowledged.

I can guarantee that I will not respond to any requests for help with assignments or homework, no matter how urgent or important they might be to you. That's what you your teachers, tutors, lecturers and professors are paid to do.

You might instead like to communicate with other users via the Google Group called "Robotics Toolbox"

http://groups.google.com.au/group/robotics-tool-box

which is a forum for discussion. You need to signup in order to post, and the signup process is moderated by me so allow a few days for this to happen. I need you to write a few words about why you want to join the list so I can distinguish you from a spammer or a web-bot.

1.3 How to obtain the Toolbox

The Robotics Toolbox is freely available from the Toolbox home page at

http://www.petercorke.com

The files are available in either gzipped tar format (.gz) or zip format (.zip). The web page requests some information from you such as your country, type of organization and application. This is just a means for me to gauge interest and to help convince my bosses (and myself) that this is a worthwhile activity.

The file `robot.pdf` is a manual that describes all functions in the Toolbox. It is auto-generated from the comments in the MATLAB® code and is fully hyperlinked: to external web sites, the table of content to functions, and the "See also" functions to each other.

A menu-driven demonstration can be invoked by the function `rtdemo`.

1.4 MATLAB version issues

The Toolbox has been tested under R2011a.

1.5 Use in teaching

This is definitely encouraged! You are free to put the PDF manual (`robot.pdf` or the web-based documentation `html/*.html` on a server for class use. If you plan to distribute paper copies of the PDF manual then every copy must include the first two pages (cover and licence).

1.6 Use in research

If the Toolbox helps you in your endeavours then I'd appreciate you citing the Toolbox when you publish. The details are

```
@ARTICLE{Corke96b,
        AUTHOR          = {P.I. Corke},
        JOURNAL         = {IEEE Robotics and Automation Magazine},
        MONTH           = mar,
        NUMBER          = {1},
```



```
        PAGES                   = {24-32},
        TITLE                   = {A Robotics Toolbox for {MATLAB}},
        VOLUME                  = {3},
        YEAR                    = {1996}
}
```

or

> "*A robotics toolbox for MATLAB*",
> P.Corke,
> IEEE Robotics and Automation Magazine,
> vol.3, pp.2432, Sept. 1996.

which is also given in electronic form in the README file.

1.7 Support, bug fixes, etc.

There is no support! This software is made freely available in the hope that you find it useful in solving whatever problems you have to hand.I am happy to correspond with people who have found genuine bugs or deficiencies but my response time can be long and I can't guarantee that I respond to your email. I am very happy to accept contributions for inclusion in future versions of the toolbox, and you will be suitably acknowledged.

I can guarantee that I will not respond to any requests for help with assignments or homework, no matter how urgent or important they might be to you. That's what you have lecturers and professors for.

You might instead like to communicate with other users via the Google Group called "Robotics Toolbox"

http://groups.google.com.au/group/robotics-tool-box

which is a forum for discussion. You need to signup in order to post, and the signup process is moderated by me so allow a few days for this to happen. I need you to write a few words about why you want to join the list so I can distinguish you from a spammer or a web-bot.

1.7.1 Other toolboxes

Also of interest might be:

- A python implementation of the Toolbox at http://code.google.com/p/robotics-toolbox-python. All core functionality of the release 8 Toolbox is present including kinematics, dynamics, Jacobians, quaternions etc. It is based on the python numpy class. The main current limitation is the lack of good 3D graphics support but people are working on this. Nevertheless this version of the toolbox is very usable and of course you don't need a MATLAB® licence to use it. Watch this space.

- Machine Vision toolbox (MVTB) for MATLAB®. This was described in an article

```
@article{Corke05d,
        Author = {P.I. Corke},
        Journal = {IEEE Robotics and Automation Magazine},
        Month = nov,
        Number = {4},
        Pages = {16-25},
        Title = {Machine Vision Toolbox},
        Volume = {12},
        Year = {2005}}
```

  and provides a very wide range of useful computer vision functions beyond the Mathwork's Image Processing Toolbox. You can obtain this from http://www.petercorke.com/vision.

1.8 Acknowledgements

Last, but not least, I have corresponded with a great many people via email since the first release of this Toolbox. Some have identified bugs and shortcomings in the documentation, and even better, some have provided bug fixes and even new modules, thankyou. See the file CONTRIB for details. I'd like to especially mention Wynand Smart for some arm robot models, Paul Pounds for the quadcopter model, and Paul Newman (Oxford) for inspiring the mobile robot code.

Chapter 2

Functions and classes

SerialLink

Serial-link robot class

r = SerialLink(**links**, **options**) is a serial-link robot object from a vector of Link objects.

r = SerialLink(**dh**, **options**) is a serial-link robot object from a table (matrix) of Denavit-Hartenberg parameters. The columns of the matrix are theta, d, alpha, a. An optional fifth column sigma indicate revolute (sigma=0, default) or prismatic (sigma=1).

Options

'name', name	set robot name property
'comment', comment	set robot comment property
'manufacturer', manuf	set robot manufacturer property
'base', base	set base transformation matrix property
'tool', tool	set tool transformation matrix property
'gravity', g	set gravity vector property
'plotopt', po	set plotting **options** property

Methods

plot	display graphical representation of robot
teach	drive the graphical robot
fkine	return forward kinematics
ikine6s	return inverse kinematics for 6-axis spherical wrist robot
ikine	return inverse kinematics using iterative method
jacob0	return Jacobian matrix in world frame
jacobn	return Jacobian matrix in tool frame
jtraj	return a joint space trajectory
dyn	show dynamic properties of **links**
isspherical	true if robot has spherical wrist
islimit	true if robot has spherical wrist
payload	add a payload in end-effector frame
coriolis	return Coriolis joint force
gravload	return gravity joint force
inertia	return joint inertia matrix
accel	return joint acceleration
fdyn	return joint motion
rne	return joint force
perturb	return SerialLink object with perturbed parameters
showlink	return SerialLink object with perturbed parameters
friction	return SerialLink object with perturbed parameters
maniplty	return SerialLink object with perturbed parameters

Properties (read/write)

links	vector of Link objects
gravity	direction of gravity [gx gy gz]
base	pose of robot's base 4×4 homog xform
tool	robot's tool transform, T6 to tool tip: 4×4 homog xform
qlim	joint limits, [qlower qupper] nx2
offset	kinematic joint coordinate offsets nx1
name	name of robot, used for graphical display
manuf	annotation, manufacturer's name
comment	annotation, general comment
plotopt	**options** for plot(robot), cell array

Object properties (read only)

n	number of joints
config	joint configuration string, eg. 'RRRRRR'
mdh	kinematic convention boolean (0=DH, 1=MDH)
islimit	joint limit boolean vector
q	joint angles from last plot operation
handle	graphics handles in object

Note

- SerialLink is a reference object.
- SerialLink objects can be used in vectors and arrays

See also

Link, DHFactor

SerialLink.SerialLink

Create a SerialLink robot object

R = SerialLink(**options**) is a null robot object with no links.

R = SerialLink(**R1**, **options**) is a deep copy of the robot object **R1**, with all the same properties.

R = SerialLink(**dh**, **options**) is a robot object with kinematics defined by the matrix **dh** which has one row per joint and each row is [theta d a alpha] and joints are assumed revolute.

R = SerialLink(**links**, **options**) is a robot object defined by a vector of Link objects.

Options

'name', name	set robot name property
'comment', comment	set robot comment property
'manufacturer', manuf	set robot manufacturer property
'base', base	set base transformation matrix property
'tool', tool	set tool transformation matrix property
'gravity', g	set gravity vector property
'plotopt', po	set plotting **options** property

Robot objects can be concatenated by:

```
R = R1 * R2;
R = SerialLink([R1 R2]);
```

which is equivalent to R2 mounted on the end of **R1**. Note that tool transform of **R1** and the base transform of R2 are lost, constant transforms cannot be represented in Denavit-Hartenberg notation.

Note

- SerialLink is a reference object, a subclass of Handle object.
- SerialLink objects can be used in vectors and arrays

See also

Link, SerialLink.plot

SerialLink.accel

Manipulator forward dynamics

qdd = R.**accel**(**q**, **qd**, **torque**) is a vector of joint accelerations that result from applying the actuator force/torque to the manipulator robot in state **q** and **qd**. If **q**, **qd**, **torque** are matrices with M rows, then **qdd** is a matrix with M rows of acceleration corresponding to the equivalent rows of **q**, **qd**, **torque**.

qdd = R.**ACCEL**(**x**) as above but **x**=[**q**,**qd**,**torque**].

Note

- Uses the method 1 of Walker and Orin to compute the forward dynamics.
- This form is useful for simulation of manipulator dynamics, in conjunction with a numerical integration function.

See also

SerialLink.rne, SerialLink, ode45

SerialLink.char

String representation of parametesrs

s = R.**char**() is a string representation of the robot parameters.

SerialLink.cinertia

Cartesian inertia matrix

m = R.**cinertia**(**q**) is the $N \times N$ Cartesian (operational space) inertia matrix which relates Cartesian force/torque to Cartesian acceleration at the joint configuration **q**, and N is the number of robot joints.

See also

SerialLink.inertia, SerialLink.rne

SerialLink.copy

Clone a robot object

r2 = R.**copy**() is a deepcopy of the object R.

SerialLink.coriolis

Coriolis matrix

C = R.**CORIOLIS**(**q**, **qd**) is the $N \times N$ Coriolis/centripetal matrix for the robot in configuration **q** and velocity **qd**, where N is the number of joints. The product **C***__qd__ is the vector of joint force/torque due to velocity coupling. The diagonal elements are due to centripetal effects and the off-diagonal elements are due to Coriolis effects. This matrix is also known as the velocity coupling matrix, since gives the disturbance forces on all joints due to velocity of any joint.

If **q** and **qd** are row vectors, the result is a row-vector of joint torques. If **q** and **qd** are matrices, each row is interpretted as a joint state vector, and the result is a matrix each row being the corresponding joint torques.

Notes

- joint friction is also a joint force proportional to velocity but it is eliminated in the computation of this value.

See also

SerialLink.rne

SerialLink.display

Display parameters

R.**display**() displays the robot parameters in human-readable form.

Notes

- this method is invoked implicitly at the command line when the result of an expression is a SerialLink object and the command has no trailing semicolon.

See also

SerialLink.char, SerialLink.dyn

SerialLink.dyn

display inertial properties

R.**dyn**() displays the inertial properties of the **SerialLink** object in a multi-line format. The properties shown are mass, centre of mass, inertia, gear ratio, motor inertia and motor friction.

See also

Link.dyn

SerialLink.fdyn

Integrate forward dynamics

[**T**,**q**,**qd**] = R.**fdyn**(**T1**, **torqfun**) integrates the dynamics of the robot over the time interval 0 to **T** and returns vectors of time TI, joint position **q** and joint velocity **qd**. The initial joint position and velocity are zero. The torque applied to the joints is computed by the user function **torqfun**:

[**ti**,**q**,**qd**] = R.**fdyn**(**T**, **torqfun**, **q0**, **qd0**) as above but allows the initial joint position and velocity to be specified.

The control torque is computed by a user defined function

TAU = **torqfun**(**T**, **q**, **qd**, ARG1, ARG2, ...)

where **q** and **qd** are the manipulator joint coordinate and velocity state respectively], and **T** is the current time.

[**T**,**q**,**qd**] = R.**fdyn**(**T1**, **torqfun**, **q0**, **qd0**, ARG1, ARG2, ...) allows optional arguments to be passed through to the user function.

Note

- This function performs poorly with non-linear joint friction, such as Coulomb friction. The R.nofriction() method can be used to set this friction to zero.
- If **torqfun** is not specified, or is given as 0 or [], then zero torque is applied to the manipulator joints.
- The builtin integration function ode45() is used.

See also

SerialLink.accel, SerialLink.nofriction, SerialLink.RNE, ode45

SerialLink.fkine

Forward kinematics

T = R.**fkine**(**q**) is the pose of the robot end-effector as a homogeneous transformation for the joint configuration **q**. For an N-axis manipulator **q** is an N-vector.

If **q** is a matrix, the M rows are interpretted as the generalized joint coordinates for a sequence of points along a trajectory. **q**(i,j) is the j'th joint parameter for the i'th trajectory point. In this case it returns a 4x4xM matrix where the last subscript is the index along the path.

Note

- The robot's base or tool transform, if present, are incorporated into the result.

See also

SerialLink.ikine, SerialLink.ikine6s

SerialLink.friction

Friction force

tau = R.**friction**(**qd**) is the vector of joint **friction** forces/torques for the robot moving with joint velocities **qd**.

The **friction** model includes viscous **friction** (linear with velocity) and Coulomb **friction** (proportional to sign(**qd**)).

See also

Link.friction

SerialLink.gravload

Gravity loading

taug = R.**gravload**(**q**) is the joint gravity loading for the robot in the joint configuration **q**. Gravitational acceleration is a property of the robot object.

If **q** is a row vector, the result is a row vector of joint torques. If **q** is a matrix, each row is interpreted as a joint configuration vector, and the result is a matrix each row being the corresponding joint torques.

taug = R.**gravload**(**q**, **grav**) is as above but the gravitational acceleration vector **grav** is given explicitly.

See also

SerialLink.rne, SerialLink.itorque, SerialLink.coriolis

SerialLink.ikine

Inverse manipulator kinematics

q = R.**ikine**(**T**) is the joint coordinates corresponding to the robot end-effector pose **T** which is a homogenenous transform.

q = R.**ikine**(**T**, **q0**) specifies the initial estimate of the joint coordinates.

q = R.**ikine**(**T**, **q0**, **m**) specifies the initial estimate of the joint coordinates and a mask matrix. For the case where the manipulator has fewer than 6 DOF the solution space has more dimensions than can be spanned by the manipulator joint coordinates. In this case the mask matrix **m** specifies the Cartesian DOF (in the wrist coordinate frame) that will be ignored in reaching a solution. The mask matrix has six elements that correspond to translation in X, Y and Z, and rotation about X, Y and Z respectively. The value should be 0 (for ignore) or 1. The number of non-zero elements should equal the number of manipulator DOF.

For example when using a 5 DOF manipulator rotation about the wrist z-axis might be unimportant in which case **m** = [1 1 1 1 1 0].

In all cases if **T** is 4x4xM it is taken as a homogeneous transform sequence and R.**ikine**() returns the joint coordinates corresponding to each of the transforms in the sequence. **q** is **m** $\times$ N where N is the number of robot joints. The initial estimate of **q** for each time step is taken as the solution from the previous time step.

Notes

- Solution is computed iteratively using the pseudo-inverse of the manipulator Jacobian.
- The inverse kinematic solution is generally not unique, and depends on the initial guess **q0** (defaults to 0).
- Such a solution is completely general, though much less efficient than specific inverse kinematic solutions derived symbolically.
- This approach allows a solution to obtained at a singularity, but the joint angles within the null space are arbitrarily assigned.

See also

SerialLink.fkine, tr2delta, SerialLink.jacob0, SerialLink.ikine6s

---

SerialLink.ikine6s

Inverse kinematics for 6-axis robot with spherical wrist

q = R.**ikine6s**(**T**) is the joint coordinates corresponding to the robot end-effector pose **T** represented by the homogenenous transform. This is a analytic solution for a 6-axis robot with a spherical wrist (such as the Puma 560).

q = R.**IKINE6S**(**T**, **config**) as above but specifies the configuration of the arm in the form of a string containing one or more of the configuration codes:

'l'	arm to the left (default)
'r'	arm to the right
'u'	elbow up (default)
'd'	elbow down
'n'	wrist not flipped (default)
'f'	wrist flipped (rotated by 180 deg)

Notes

- The inverse kinematic solution is generally not unique, and depends on the configuration string.

Reference

Inverse kinematics for a PUMA 560 based on the equations by Paul and Zhang From The International Journal of Robotics Research Vol. 5, No. 2, Summer 1986, p. 32-44

Author

Robert Biro with Gary Von McMurray, GTRI/ATRP/IIMB, Georgia Institute of Technology 2/13/95

See also

SerialLink.FKINE, SerialLink.IKINE

SerialLink.inertia

Manipulator inertia matrix

i = R.**inertia**(**q**) is the $N \times N$ symmetric joint **inertia** matrix which relates joint torque to joint acceleration for the robot at joint configuration **q**. The diagonal elements **i**(j,j) are the **inertia** seen by joint actuator j. The off-diagonal elements are coupling inertias that relate acceleration on joint i to force/torque on joint j.

See also

SerialLink.RNE, SerialLink.CINERTIA, SerialLink.ITORQUE

SerialLink.islimit

Joint limit test

v = R.**ISLIMIT**(**q**) is a vector of boolean values, one per joint, false (0) if **q**(i) is within the joint limits, else true (1).

SerialLink.isspherical

Test for spherical wrist

R.**isspherical**() is true if the robot has a spherical wrist, that is, the last 3 axes intersect at a point.

See also

SerialLink.ikine6s

SerialLink.itorque

Inertia torque

taui = R.**itorque**(**q**, **qdd**) is the inertia force/torque N-vector at the specified joint configuration **q** and acceleration **qdd**, that is, **taui** = INERTIA(**q**)***qdd**.

If **q** and **qdd** are row vectors, the result is a row vector of joint torques. If **q** and **qdd** are matrices, each row is interpretted as a joint state vector, and the result is a matrix each row being the corresponding joint torques.

Note

- If the robot model contains non-zero motor inertia then this will included in the result.

See also

SerialLink.rne, SerialLink.inertia

SerialLink.jacob0

Jacobian in world coordinates

j0 = R.**jacob0**(**q**, **options**) is a $6 \times N$ Jacobian matrix for the robot in pose **q**. The manipulator Jacobian matrix maps joint velocity to end-effector spatial velocity V = **j0***QD expressed in the world-coordinate frame.

Options

'rpy'	Compute analytical Jacobian with rotation rate in terms of roll-pitch-yaw angles
'eul'	Compute analytical Jacobian with rotation rates in terms of Euler angles
'trans'	Return translational submatrix of Jacobian
'rot'	Return rotational submatrix of Jacobian

Note

- the Jacobian is computed in the world frame and transformed to the end-effector frame.

See also

SerialLink.jacobn, deltatr, tr2delta

SerialLink.jacob_dot

Hessian in end-effector frame

jdq = R.jacob_dot(**q**, **qd**) is the product of the Hessian, derivative of the Jacobian, and the joint rates.

Notes

- useful for operational space control
- not yet tested/debugged.

See also

: SerialLink.jacob0, diff2tr, tr2diff

SerialLink.jacobn

Jacobian in end-effector frame

jn = R.**jacobn**(**q**, **options**) is a $6 \times N$ Jacobian matrix for the robot in pose **q**. The manipulator Jacobian matrix maps joint velocity to end-effector spatial velocity V = J0*QD in the end-effector frame.

Options

'trans'	Return translational submatrix of Jacobian
'rot'	Return rotational submatrix of Jacobian

Reference

Paul, Shimano, Mayer, Differential Kinematic Control Equations for Simple Manipulators, IEEE SMC 11(6) 1981, pp. 456-460

See also

SerialLink.jacob0, delta2tr, tr2delta

SerialLink.jtraj

Create joint space trajectory

q = R.**jtraj**(**T0**, **tf**, **m**) is a joint space trajectory where the joint coordinates reflect motion from end-effector pose **T0** to **tf** in **m** steps with default zero boundary conditions for velocity and acceleration. The trajectory **q** is an **m** $\times$ N matrix, with one row per time step, and one column per joint, where N is the number of robot joints.

Note

- requires solution of inverse kinematics. R.ikine6s() is used if appropriate, else R.ikine(). Additional trailing arguments to R.**jtraj**() are passed as trailing arugments to the these functions.

See also

jtraj, SerialLink.ikine, SerialLink.ikine6s

SerialLink.maniplty

Manipulability measure

m = R.**maniplty**(**q**, **options**) is the manipulability index measure for the robot at the joint configuration **q**. It indicates dexterity, how isotropic the robot's motion is with respect to the 6 degrees of Cartesian motion. The measure is low when the manipulator is close to a singularity. If **q** is a matrix **m** is a column vector of manipulability indices for each pose specified by a row of **q**.

Two measures can be selected:

- Yoshikawa's manipulability measure is based on the shape of the velocity ellipsoid and depends only on kinematic parameters.
- Asada's manipulability measure is based on the shape of the acceleration ellipsoid which in turn is a function of the Cartesian inertia matrix and the dynamic parameters. The scalar measure computed here is the ratio of the smallest/largest ellipsoid axis. Ideally the ellipsoid would be spherical, giving a ratio of 1, but in practice will be less than 1.

Options

'T'	compute manipulability for just transational motion
'R'	compute manipulability for just rotational motion
'yoshikawa'	use Asada algorithm (default)
'asada'	use Asada algorithm

Notes

- by default the measure includes rotational and translational dexterity, but this involves adding different units. It can be more useful to look at the translational and rotational manipulability separately.

See also

SerialLink.inertia, SerialLink.jacob0

SerialLink.mtimes

Join robots

R = R1 * R2 is a robot object that is equivalent to mounting robot R2 on the end of robot R1.

SerialLink.nofriction

Remove friction

rnf = R.**nofriction**() is a robot object with the same parameters as R but with non-linear (Couolmb) friction coefficients set to zero.

rnf = R.**nofriction**('all') as above but all friction coefficients set to zero.

Notes:

- Non-linear (Coulomb) friction can cause numerical problems when integrating the equations of motion (R.fdyn).
- The resulting robot object has its name string modified by prepending 'NF/'.

See also

SerialLink.fdyn, Link.nofriction

SerialLink.payload

Add payload to end of manipulator

R.**payload**(**m**, **p**) adds a **payload** with point mass **m** at position **p** in the end-effector coordinate frame.

See also

SerialLink.ikine6s

SerialLink.perturb

Perturb robot parameters

rp = R.**perturb**(**p**) is a new robot object in which the dynamic parameters (link mass and inertia) have been perturbed. The perturbation is multiplicative so that values are multiplied by random numbers in the interval (1-**p**) to (1+**p**). The name string of the perturbed robot is prefixed by '**p/**'.

Useful for investigating the robustness of various model-based control schemes. For example to vary parameters in the range +/- 10 percent is:

```
r2 = p560.perturb(0.1);
```

SerialLink.plot

Graphical display and animation

R.**plot**(**q**, **options**) displays a graphical animation of a robot based on the kinematic model. A stick figure polyline joins the origins of the link coordinate frames. The robot is displayed at the joint angle **q**, or if a matrix it is animated as the robot moves along the trajectory.

The graphical robot object holds a copy of the robot object and the graphical element is tagged with the robot's name (.name property). This state also holds the last joint configuration which can be retrieved, see PLOT(robot) below.

Figure behaviour

If no robot of this name is currently displayed then a robot will be drawn in the current figure. If hold is enabled (hold on) then the robot will be added to the current figure.

If the robot already exists then that graphical model will be found and moved.

Multiple views of the same robot

If one or more plots of this robot already exist then these will all be moved according to the argument **q**. All robots in all windows with the same name will be moved.

Multiple robots in the same figure

Multiple robots can be displayed in the same **plot**, by using "hold on" before calls to **plot**(robot).

Graphical robot state

The configuration of the robot as displayed is stored in the **SerialLink** object and can be accessed by the read only object property R.q.

Graphical annotations and options

The robot is displayed as a basic stick figure robot with annotations such as:

- shadow on the floor
- XYZ wrist axes and labels
- joint cylinders and axes

which are controlled by **options**.

The size of the annotations is determined using a simple heuristic from the workspace dimensions. This dimension can be changed by setting the multiplicative scale factor using the 'mag' option.

Options

'workspace', W	size of robot 3D workspace, W = [xmn, xmx ymn ymx zmn zmx]
'delay', d	delay betwen frames for animation (s)
'cylinder', C	color for joint cylinders, C=[r g b]
'mag', scale	annotation scale factor
'perspective'—'ortho'	type of camera view
'raise'—'noraise'	controls autoraise of current figure on **plot**
'render'—'norender'	controls shaded rendering after drawing
'loop'—'noloop'	controls endless loop mode
'base'—'nobase'	controls display of base 'pedestal'
'wrist'—'nowrist'	controls display of wrist
'shadow'—'noshadow'	controls display of shadow
'name'—'noname'	display the robot's name
'xyz'—'noa'	wrist axis label
'jaxes'—'nojaxes'	control display of joint axes
'joints'—'nojoints'	controls display of joints

The **options** come from 3 sources and are processed in order:

- Cell array of **options** returned by the function PLOTBOTOPT.
- Cell array of **options** given by the 'plotopt' option when creating the SerialLink object.
- List of arguments in the command line.

See also

plotbotopt, SerialLink.fkine

SerialLink.rne

Inverse dynamics

tau = R.**rne**(**q**, **qd**, **qdd**) is the joint torque required for the robot R to achieve the specified joint position **q**, velocity **qd** and acceleration **qdd**.

tau = R.**rne**(**q**, **qd**, **qdd**, **grav**) as above but overriding the gravitational acceleration vector in the robot object R.

tau = R.**rne**(**q**, **qd**, **qdd**, **grav**, **fext**) as above but specifying a wrench acting on the end of the manipulator which is a 6-vector [Fx Fy Fz Mx My Mz].

tau = R.**rne**(**x**) as above where **x**=[**q**,**qd**,**qdd**].

tau = R.**rne**(**x**, **grav**) as above but overriding the gravitational acceleration vector in the robot object R.

tau = R.**rne**(**x**, **grav**, **fext**) as above but specifying a wrench acting on the end of the manipulator which is a 6-vector [Fx Fy Fz Mx My Mz].

If **q**,**qd** and **qdd**, or **x** are matrices with M rows representing a trajectory then **tau** is an $M \times N$ matrix with rows corresponding to each trajectory state.

Notes:

- The torque computed also contains a contribution due to armature inertia.
- **rne** can be either an M-file or a MEX-file. See the manual for details on how to configure the MEX-file. The M-file is a wrapper which calls either **rne**_DH or **rne**_MDH depending on the kinematic conventions used by the robot object.

See also

SerialLink.accel, SerialLink.gravload, SerialLink.inertia

SerialLink.showlink

Show parameters of all links

R.**showlink**() shows details of all link parameters for the robot object, including inertial parameters.

See also

Link.showlink, Link

SerialLink.teach

Graphical teach pendant

R.**teach**() drive a graphical robot by means of a graphical slider panel. If no graphical robot exists one is created in a new window. Otherwise all current instanes of the graphical robots are driven.

R.**teach**(**q**) specifies the initial joint angle, otherwise it is taken from one of the existing graphical robots.

See also

SerialLink.plot

Bug2

Bug navigation class

A concrete subclass of Navigation that implements the bug2 navigation algorithm. This is a simple automaton that performs local planning, that is, it can only sense the immediate presence of an obstacle.

Methods

path	Compute a path from start to goal
visualize	Display the occupancy grid
display	Display the state/parameters in human readable form
char	Convert the state/parameters to human readable form

Example

```
load map1
bug = Bug2(map);
bug.goal = [50; 35];
bug.path([20; 10]);
```

See also

Navigation, DXform, Dstar, PRM

Bug2.Bug2

bug2 navigation object constructor

b = Bug2(**map**) is a bug2 navigation object, and **map** is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

b = Bug2(**map**, **goal**) as above but specify the goal point.

See also

Navigation.Navigation

DHFactor

Simplify symbolic link transform expressions

f = **dhfactor**(**s**) is an object that encodes the kinematic model of a robot provided by a string **s** that represents a chain of elementary transforms from the robot's base to its tool tip. The chain of elementary rotations and translations is symbolically factored into a sequence of link transforms described by DH parameters.

For example:

```
s = 'Rz(q1).Rx(q2).Ty(L1).Rx(q3).Tz(L2)';
```

indicates a rotation of q1 about the z-axis, then rotation of q2 about the x-axis, translation of L1 about the y-axis, rotation of q3 about the x-axis and translation of L2 along the z-axis.

Methods

display	shows the simplified version in terms of Denavit-Hartenberg parameters
base	shows the base transform
tool	shows the tool transform
command	returns a string that could be passed to the SerialLink() object constructor to generate a robot with these kinematics.

Example

```
>> s = 'Rz(q1).Rx(q2).Ty(L1).Rx(q3).Tz(L2)';
>> dh = DHFactor(s);
>> dh
DH(q1+90, 0, 0, +90).DH(q2, L1, 0, 0).DH(q3-90, L2, 0, 0).Rz(+90).Rx(-90).Rz(-90)
>> r = eval( dh.command() );
```

Notes

- Variables starting with q are assumed to be joint coordinates
- Variables starting with L are length constants.
- implemented in Java

See also

SerialLink

DXform

Distance transform navigation class

A concrete subclass of Navigation that implements the distance transform navigation algorithm. This provides minimum distance paths.

Methods

plan	Compute the cost map given a goal and map
path	Compute a path to the goal
visualize	Display the obstacle map
display	Print the parameters in human readable form
char	Convert the parameters to a human readable string

Properties

metric	The distance metric, can be 'euclidean' (default) or 'cityblock'
distance	The distance transform of the occupancy grid

Example

```
load map1
dx = DXform(map);
dx.plan(goal)
dx.path(start)
```

See also

Navigation, Dstar, PRM, distancexform

DXform.DXform

Distance transform navigation constructor

dx = **DXform**(**map**) is a distance transform navigation object, and **map** is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

ds = **Dstar**(**map**, **goal**) as above but specify the goal point.

See also

Navigation.Navigation

DXform.char

Convert navigation object to string

DX.**char**() is a string representing the state of the navigation object in human-readable form.

See also

DXform.display

DXform.plan

Plan path to goal

DX.**plan**() updates DX with a costmap of distance to the goal from every non-obstacle point in the map. The goal is as specified to the constructor.

DX.**plan**(**goal**) as above but uses the specified goal

DX.**plan**(**goal**, **s**) as above but displays the evolution of the costmap, with one iteration displayed every **s** seconds.

DXform.setgoal

the imorph primitive we need to set the target pixel to 0,

obstacles to NaN and the rest to Inf. invoked by superclass constructor

---

DXform.visualize

Visualize navigation environment

DX.**visualize**() displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

DX.**visualize**(**p**) as above but also overlays the points **p** in the path points which is an $N \times 2$ matrix.

See also

Navigation.visualize

Dstar

D* navigation class

A concrete subclass of Navigation that implements the distance transform navigation algorithm. This provides minimum distance paths and facilitates incremental replanning.

Methods

plan	Compute the cost map given a goal and map
path	Compute a path to the goal
visualize	Display the obstacle map
display	Print the parameters in human readable form
char	Convert the parameters to a human readable string
modify_cost	Modify the costmap
costmap_get	Return the current costmap

Example

```
load map1
ds = Dstar(map);
ds.plan(goal)
ds.path(start)
```

See also

Navigation, DXform, PRM

Dstar.Dstar

D* navigation constructor

ds = **Dstar**(**map**) is a D* navigation object, and **map** is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).. The occupancy grid is coverted to a costmap with a unit cost for traversing a cell.

ds = **Dstar**(**map**, **goal**) as above but specify the goal point.

See also

Navigation.Navigation

Dstar.char

Convert navigation object to string

DS.**char**() is a string representing the state of the navigation object in human-readable form.

See also

Dstar.display

Dstar.costmap_get

Get the current costmap

C = DS.costmap_get() returns the current costmap.

Dstar.modify_cost

Modify cost map

DS.modify_cost(**p**, **new**) modifies the cost map at **p**=[X,Y] to have the value **new**.

After one or more point costs have been updated the path should be replanned by calling DS.plan().

Dstar.plan

Plan path to goal

DS.**plan**() updates DS with a costmap of distance to the goal from every non-obstacle point in the map. The goal is as specified to the constructor.

DS.**plan**(**goal**) as above but uses the specified goal.

Note

- if a path has already been planned, but the costmap was modified, then reinvoking this method will replan, incrementally updating the **plan** at lower cost than a full replan.

Dstar.reset

Reset the planner

DS.**reset**() resets the D* planner. The next instantiation of DS.plan() will perform a global replan.

Dstar.visualize

Visualize navigation environment

DS.**visualize**() displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

DS.**visualize**(**p**) as above but also overlays the points **p** in the path points which is an $N \times 2$ matrix.

See also

Navigation.visualize

EKF

Extended Kalman Filter for vehicle pose and map estimation

This class can be used for:

- dead reckoning localization
- map-based localization
- map making
- simultaneous localization and mapping

It is used in conjunction with:

- a kinematic vehicle model that provides odometry output, represented by a Vehicle object.
- The vehicle must be driven within the area of the map and this is achieved by connecting it to a Driver object.
- a map containing the position of a number of landmarks, a Map object
- a sensor that returns measurements about landmarks relative to the vehicle's location.

The **EKF** object updates its state at each time step, and invokes the state update methods of the Vehicle. The complete history of estimated state and covariance is stored within the **EKF** object.

Methods

run	run the filter
plot_xy	return/plot the actual path of the vehicle
plot_P	return/plot the estimate covariance
plot_map	plot feature points and confidence limits
plot_ellipse	plot path with covariance ellipses
display	print the filter state in human readable form
char	convert the filter state to human readable string

Properties

x_est	estimated state
P	estimated covariance
V_est	estimated odometry covariance
W_est	estimated sensor covariance
features	map book keeping, maps sensor feature id to filter state
robot	reference to the robot object
sensor	reference to the sensor object
history	vector of structs that hold the detailed information from each time step

Vehicle position estimation

Create a vehicle with odometry covariance V, add a driver to it, create a Kalman filter with estimated covariance V_est and initial state covariance P0, then run the filter for N time steps.

```
veh = Vehicle(V);
veh.add_driver( RandomPath(20, 2) );
ekf = EKF(veh, V_est, P0);
ekf.run(N);
```

Vehicle map based localization

Create a vehicle with odometry covariance V, add a driver to it, create a map with 20 point features, create a sensor that uses the map and vehicle state to estimate feature range and bearing with covariance W, the Kalman filter with estimated covariances V_est and W_est and initial vehicle state covariance P0, then run the filter for N time steps.

```
veh = Vehicle(V);
veh.add_driver( RandomPath(20, 2) );
map = Map(20);
sensor = RangeBearingSensor(veh, map, W);
ekf = EKF(veh, V_est, P0, sensor, W_est, map);
ekf.run(N);
```

Vehicle-based map making

Create a vehicle with odometry covariance V, add a driver to it, create a sensor that uses the map and vehicle state to estimate feature range and bearing with covariance W, the Kalman filter with estimated sensor covariance W_est and a "perfect" vehicle (no covariance), then run the filter for N time steps.

```
veh = Vehicle(V);
veh.add_driver( RandomPath(20, 2) );
sensor = RangeBearingSensor(veh, map, W);
ekf = EKF(veh, [], [], sensor, W_est, []);
ekf.run(N);
```

Simultaneous localization and mapping (SLAM)

Create a vehicle with odometry covariance V, add a driver to it, create a map with 20 point features, create a sensor that uses the map and vehicle state to estimate feature range and bearing with covariance W, the Kalman filter with estimated covariances V_est and W_est and initial state covariance P0, then run the filter for N time steps to estimate

the vehicle state at each time step and the map.% veh = Vehicle(V);

```
veh.add_driver( RandomPath(20, 2) );
map = Map(20);
sensor = RangeBearingSensor(veh, map, W);
```



```
ekf = EKF(veh, V_est, P0, sensor, W, []);
ekf.run(N);
```

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

Vehicle, RandomPath, RangeBearingSensor, Map, ParticleFilter

EKF.EKF

EKF object constructor

E = **EKF**(**vehicle**, **vest**, **p0**) is an **EKF** that estimates the state of the **vehicle** with estimated odometry covariance **vest** (2×2) and initial covariance (3×3).

E = **EKF**(**vehicle**, **vest**, **p0**, **sensor**, **west**, **map**) as above but uses information from a **vehicle** mounted sensor, estimated sensor covariance **west** and a **map**.

If **map** is [] then it will be estimated.

If **vest** and **p0** are [] the vehicle is assumed error free and the filter will estimate the landmark positions (map).

If **vest** and **p0** are finite the filter will estimate the vehicle pose and the landmark positions (map).

Notes

- EKF subclasses Handle, so it is a reference object.

See also

Vehicle, Sensor, RangeBearingSensor, Map

EKF.char

Convert EKF object to string

E.**char**() is a string representing the state of the **EKF** object in human-readable form.

EKF.display

Display status of EKF object

E.**display**() **display** the state of the **EKF** object in human-readable form.

Notes

- this method is invoked implicitly at the command line when the result of an expression is a EKF object and the command has no trailing semicolon.

See also

EKF.char

EKF.plot_P

Plot covariance magnitude

E.plot_P() plots the estimated covariance magnitude against time step.

E.plot_P(**ls**) as above but the optional line style arguments **ls** are passed to plot.

m = E.plot_P() returns the estimated covariance magnitude at all time steps as a vector.

EKF.plot_ellipse

Plot vehicle covariance as an ellipse

E.plot_ellipse(**i**) overlay the current plot with the estimated vehicle position covariance ellipses for every **i**'th time step.

E.plot_ellipse() as above but **i**=20.

E.plot_ellipse(**i**, **ls**) as above but pass line style arguments **ls** to plot_ellipse.

See also

plot_ellipse

EKF.plot_map

Plot landmarks

E.plot_map(**i**) overlay the current plot with the estimated landmark position (a +-marker) and a covariance ellipses for every **i**'th time step.

E.plot_map() as above but **i**=20.

E.plot_map(**i**, **ls**) as above but pass line style arguments **ls** to plot_ellipse.

See also

plot_ellipse

EKF.plot_xy

Plot vehicle position

E.plot_xy() plot the estimated vehicle path in the xy-plane.

E.plot_xy(**ls**) as above but the optional line style arguments **ls** are passed to plot.

EKF.run

Run the EKF

E.**run**(**n**) **run** the filter for **n** time steps.

Notes

- all previously estimated states and estimation history is cleared.

Link

Robot manipulator Link class

A **Link** object holds all information related to a robot link such as kinematics parameteres, rigid-body inertial parameters, motor and transmission parameters.

L = **Link**([theta d a alpha]) is a link object with the specified kinematic parameters theta, d, a and alpha.

Methods

A	return link transform (A) matrix
RP	return joint type: 'R' or 'P'
friction	return friction force
nofriction	return Link object with friction parameters set to zero
dyn	display link dynamic parameters
islimit	true if joint exceeds soft limit
isrevolute	true if joint is revolute
isprismatic	true if joint is prismatic
nofriction	remove joint friction
display	print the link parameters in human readable form
char	convert the link parameters to human readable string

Properties (read/write)

alpha	kinematic: link twist
a	kinematic: link twist
theta	kinematic: link twist
d	kinematic: link twist
sigma	kinematic: 0 if revolute, 1 if prismatic
mdh	kinematic: 0 if standard D&H, else 1

offset kinematic: joint variable offset

qlim	kinematic: joint variable limits [min max]
m	dynamic: link mass
r	dynamic: link COG wrt link coordinate frame 3×1
I	dynamic: link inertia matrix, symmetric 3×3, about link COG.
B	dynamic: link viscous friction (motor referred)
Tc	dynamic: link Coulomb friction
G	actuator: gear ratio
J	actuator: motor inertia (motor referred)

Notes

- this is reference class object
- Link objects can be used in vectors and arrays

See also

SerialLink, Link.Link

Link.A

Link transform matrix

T = L.A(**q**) is the 4×4 link homogeneous transformation matrix corresponding to the link variable **q** which is either theta (revolute) or d (prismatic).

Notes

- For a revolute joint the theta parameter of the link is ignored, and **q** used instead.
- For a prismatic joint the d parameter of the link is ignored, and **q** used instead.
- The link offset parameter is added to **q** before computation of the transformation matrix.

Link.Link

Create robot link object

This is class constructor function which has several call signatures.

L = **Link**() is a **Link** object with default parameters.

L = **Link**(**l1**) is a **Link** object that is a deep copy of the object **l1**.

L = **Link**(**dh**, **options**) is a link object formed from the kinematic parameter vector:

- **dh** = [theta d a alpha sigma offset] where offset is a constant added to the joint angle variable before forward kinematics and is useful if you want the robot to adopt a 'sensible' pose for zero joint angle configuration.
- **dh** = [theta d a alpha sigma] where sigma=0 for a revolute and 1 for a prismatic joint, offset is zero.
- **dh** = [theta d a alpha], joint is assumed revolute and offset is zero.

Options

'standard' for standard D&H parameters (default).
'modified' for modified D&H parameters.

Notes:

- Link object is a reference object, a subclass of Handle object.
- Link objects can be used in vectors and arrays
- the parameter theta or d is unused in a revolute or prismatic joint respectively, it is simply a placeholder for the joint variable passed to **L**.A()

- the link dynamic (inertial and motor) parameters are all set to zero. These must be set by explicitly assigning the object properties: m, r, I, Jm, B, Tc, G.

Link.RP

Joint type

c = L.**RP**() is a character 'R' or 'P' depending on whether joint is revolute or prismatic respectively. If L is a vector of **Link** objects return a string of characters in joint order.

Link.char

String representation of parameters

s = L.**char**() is a string showing link parameters in compact single line format. If L is a vector of **Link** objects return a string with one line per **Link**.

See also

Link.display

Link.display

Display parameters

L.**display**() **display** link parameters in compact single line format. If L is a vector of **Link** objects **display** one line per element.

Notes

- this method is invoked implicitly at the command line when the result of an expression is a Link object and the command has no trailing semicolon.

See also

Link.char, Link.dyn, SerialLink.showlink

Link.dyn

display the inertial properties of link

L.**dyn**() displays the inertial properties of the link object in a multi-line format. The properties shown are mass, centre of mass, inertia, friction, gear ratio and motor properties.

If L is a vector of **Link** objects show properties for each element.

Link.friction

Joint friction force

f = L.**friction**(**qd**) is the joint **friction** force/torque for link velocity **qd**

Link.islimit

Test joint limits

L.**islimit**(**q**) is true (1) if **q** is outside the soft limits set for this joint.

Link.isprismatic

Test if joint is prismatic

L.**isprismatic**() is true (1) if joint is prismatic.

See also

Link.isrevolute

Link.isrevolute

Test if joint is revolute

L.**isrevolute**() is true (1) if joint is revolute.

See also

Link.isprismatic

Link.nofriction

Remove friction

ln = L.**nofriction**() is a link object with the same parameters as L except nonlinear (Coulomb) friction parameter is zero.

ln = L.**nofriction**('all') is a link object with the same parameters as L except all friction parameters are zero.

Link.set.I

Set link inertia

L.I = [Ixx Iyy Izz] set **Link** inertia to a diagonal matrix.

L.I = [Ixx Iyy Izz Ixy Iyz Ixz] set **Link** inertia to a symmetric matrix with specified intertia and product of intertia elements.

L.I = M set **Link** inertia matrix to 3×3 matrix M (which must be symmetric).

Link.set.Tc

Set Coulomb friction

L.Tc = F set Coulomb friction parameters to [-F F], for a symmetric Coulomb friction model.

L.Tc = [FP FM] set Coulomb friction to [FP FM], for an asymmetric Coulomb friction model. FP>0 and FM<0.

See also

Link.friction

Map

Map of planar point features

m = **Map**(**n**, **dim**) returns a **Map** object that represents **n** random point features in a planar region bounded by +/-**dim** in the x- and y-directions.

Methods

plot	Plot the feature map
feature	Return a specified map feature
display	Display map parameters in human readable form
char	Convert map parameters to human readable string

Properties

map	Matrix of map feature coordinates $2 \times$ **n**
dim	The dimensions of the map region x,y in [-dim,dim]
nfeatures	The number of map features **n**

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

RangeBearingSensor, EKF

Map.Map

Map of point feature landmarks

m = **Map**(**n**, **dim**) is a **Map** object that represents **n** random point features in a planar region bounded by +/-**dim** in the x- and y-directions.

Map.char

Convert vehicle parameters and state to a string

s = M.**char**() is a string showing map parameters in a compact human readable format.

Map.display

Display map parameters

M.**display**() **display** map parameters in a compact human readable form.

Notes

- this method is invoked implicitly at the command line when the result of an expression is a Map object and the command has no trailing semicolon.

See also

map.char

Map.feature

Return the specified map feature

f = M.**feature**(**k**) is the 2×1 coordinate vector of the **k**'th **feature**.

Map.plot

Plot the feature map

M.**plot**() plots the feature map in the current figure, as a square region with dimensions given by the M.dim property. Each feature is marked by a black diamond.

M.**plot**(**ls**) plots the feature map as above, but the arguments **ls** are passed to **plot** and override the default marker style.

Notes

- The **plot** is left with HOLD ON.

Map.verbosity

Set verbosity

M.**verbosity**(**v**) set **verbosity** to **v**, where 0 is silent and greater values display more information.

Navigation

Navigation superclass

An abstract superclass for implementing navigation classes.

nav = **Navigation**(**occgrid**, **options**) is an instance of the **Navigation** object.

Methods

visualize	display the occupancy grid
plan	plan a path to goal
path	return/animate a path from start to goal
display	print the parameters in human readable form
char	convert the parameters to a human readable string

Properties (read only)

occgrid	occupancy grid representing the navigation environment
goal	goal coordinate

Methods to be provided in subclass

goal_set	set the goal
world_set	set the occupancy grid
navigate_init	

plan	generate a plan for motion to goal
next	returns coordinate of next point on path

Notes

- subclasses the Matlab handle class which means that pass by reference semantics apply.

See also

Dstar, dxform, PRM, RRT

Navigation.Navigation

Create a Navigation object

n = Navigation(**occgrid**, **options**) is a Navigation object that holds an occupancy grid **occgrid**. A number of **options** can be be passed.

Options

'navhook', F	Specify a function to be called at every step of path
'seed', s	Specify an initial random number seed
'goal', g	Specify the goal point
'verbose'	Display debugging information

Navigation.char

Convert navigation object to string

N.**char**() is a string representing the state of the navigation object in human-readable form.

Navigation.display

Display status of navigation object

N.**display**() **display** the state of the navigation object in human-readable form.

Notes

- this method is invoked implicitly at the command line when the result of an expression is a Navigation object and the command has no trailing semicolon.

See also

Navigation.char

Navigation.path

Follow path from start to goal

N.**path**(**start**) animates the robot moving from **start** to the goal (which is a property of the object).

N.**path**() display the occupancy grid, prompt the user to click a start location, then compute a **path** from this point to the goal (which is a property of the object).

x = N.**path**(**start**) returns the **path** from **start** to the goal (which is a property of the object).

The method performs the following steps:

- get start position interactively if not given
- initialized navigation, invoke method N.navigate_init()
- visualize the environment, invoke method N.visualize()
- iterate on the next() method of the subclass

See also

Navigation.visualize, Navigation.goal

Navigation.verbosity

Set verbosity

N.**verbosity**(**v**) set **verbosity** to **v**, where 0 is silent and greater values display more information.

Navigation.visualize

Visualize navigation environment

N.**visualize**() displays the occupancy grid in a new figure.

N.**visualize**(**p**) displays the occupancy grid in a new figure, and shows the path points **p** which is an $N \times 2$ matrix.

Options

'goal'	Superimpose the goal position if set
'distance', D	Display a distance field D behind the obstacle map. D is a matrix of the same size as the occupancy grid.

PGraph

Simple graph class

g = PGraph()	create a 2D, planar, undirected graph
g = PGraph(n)	create an n-d, undirected graph

Graphs

- are undirected
- are symmetric cost edges (A to B is same cost as B to A)
- are embedded in coordinate system
- have no loops (edges from A to A)
- vertices are represented by integer ids, vid
- edges are represented by integer ids, eid

Graph connectivity is maintained by a labeling algorithm and this is updated every time an edge is added.

Methods

Constructing the graph

g.add_node(coord)	add vertex, return vid
g.add_node(coord, v)	add vertex and edge to v, return vid
g.add_edge(v1, v2)	add edge from v1 to v2, return eid
g.clear()	remove all nodes and edges from the graph

Information from graph

g.edges(e)	return vid for edge
g.cost(e)	return cost for edge list
g.coord(v)	return coordinate of node v
g.neighbours(v)	return vid for edge
g.component(v)	return component id for vertex
g.connectivity()	return number of edges for all nodes
g.plot()	set goal vertex for path planning
g.pick()	return vertex id closest to picked point
char(g)	display summary info about the graph

Planning paths through the graph

g.goal(v)	set goal vertex, and plan paths
g.next(v)	return d of neighbour of v closest to goal
g.path(v)	return list of nodes from v to goal

Graph and world points

g.distance(v1, v2)	distance between v1 and v2 as the crow flies
g.closest(coord)	return vertex closest to coord
g.distances(coord)	return sorted distances from coord and vertices

To change the distance metric create a subclass of **PGraph** and override the method distance_metric().

Object properties (read/write)

g.n	number of nodes

PGraph.PGraph

Graph class constructor

g = **PGraph**(**d**, **options**) returns a graph object embedded in **d** dimensions.

Options

'distance', M	Use the distance metric M for path planning
'verbose'	Specify verbose operation

Note

- The distance metric is either 'Euclidean' or 'SE2' which is the sum of the squares of the difference in position and angle modulo 2pi.

PGraph.add_edge

Add an edge to the graph

E = G.add_edge(**v1**, **v2**) add an edge between nodes with id **v1** and **v2**, and returns the edge id **E**.

E = G.add_edge(**v1**, **v2**, **C**) add an edge between nodes with id **v1** and **v2** with cost **C**.

PGraph.add_node

Add a node to the graph

v = G.add_node(**x**) adds a node with coordinate **x**, where **x** is $D \times 1$, and returns the node id **v**.

v = G.add_node(**x**, **v**) adds a node with coordinate **x** and connected to node **v** by an edge.

v = G.add_node(**x**, **v**, **C**) adds a node with coordinate **x** and connected to node **v** by an edge with cost **C**.

PGraph.char

Convert graph to string

s = G.**char**() returns a compact human readable representation of the state of the graph including the number of vertices, edges and components.

PGraph.clear

Clear the graph

G.**CLEAR**() removes all nodes and edges.

PGraph.closest

Find closest node

v = G.**closest**(**x**) return id of node geometrically **closest** to coordinate **x**.

[**v**,**d**] = G.**CLOSEST**(**x**) return id of node geometrically closest to coordinate **x**, and the distance **d**.

PGraph.connectivity

Graph connectivity

C = G.**connectivity**() returns the total number of edges in the graph.

PGraph.coord

Coordinate of node

x = G.**coord**(**v**) return coordinate vector, $D \times 1$, of node id **v**.

PGraph.cost

Cost of edge

C = G.**cost**(**E**) return **cost** of edge id **E**.

PGraph.display

Display state of the graph

G.**display**() displays a compact human readable representation of the state of the graph including the number of vertices, edges and components.

See also

PGraph.char

PGraph.distance

Distance between nodes

d = G.**distance**(**v1**, **v2**) return the geometric **distance** between the nodes with id **v1** and **v2**.

PGraph.distances

distance to all nodes

d = G.**distances**(**v**) returns vector of geometric distance from node id **v** to every other node (including **v**) sorted into increasing order by **d**.

[**d**,**w**] = G.**distances**(**v**) returns vector of geometric distance from node id **v** to every other node (including **v**) sorted into increasing order by **d** where elements of **w** are the corresponding node id.

PGraph.edges

Find edges given vertex

E = G.**edges**(**v**) return the id of all **edges** from node id **v**.

PGraph.goal

Set goal node

G.**goal**(**vg**) for least-cost path through graph set the **goal** node. The cost of reaching every node in the graph connected to **vg** is computed.

See also

PGraph.path

cost is total distance from **goal**

PGraph.neighbours

Neighbours of a node

n = G.**neighbours**(**v**) return a vector of ids for all nodes which are directly connected **neighbours** of node id **v**.

[**n**,**C**] = G.**neighbours**(**v**) return a vector **n** of ids for all nodes which are directly connected **neighbours** of node id **v**. The elements of **C** are the edge costs of the paths to the corresponding node ids in **n**.

PGraph.next

Find next node toward goal

v = G.**next**(**vs**) return the id of a node connected to node id **vs** that is closer to the goal.

See also

PGraph.goal, PGraph.path

PGraph.path

Find path to goal node

p = G.**path**(**vs**) return a vector of node ids that form a **path** from the starting node **vs** to the previously specified goal. The **path** includes the start and goal node id.

See also

PGraph.goal

PGraph.pick

Graphically select a node

v = G.**pick**() returns the id of the node closest to the point clicked by user on a plot of the graph.

See also

PGraph.plot

PGraph.plot

Plot the graph

G.**plot**(**opt**) **plot** the graph in the current figure. Nodes are shown as colored circles.

Options

'labels'	Display node id (default false)
'edges'	Display edges (default true)
'edgelabels'	Display edge id (default false)
'MarkerSize', S	Size of node circle
'MarkerFaceColor', C	Node circle color
'MarkerEdgeColor', C	Node circle edge color
'componentcolor'	Node color is a function of graph component

PGraph.showComponent

t

G.**showcomponent**(**C**) plots the nodes that belong to graph component **C**.

PGraph.showVertex

Highlight a vertex

G.**showVertex**(**v**) highlights the vertex **v** with a yellow marker.

PGraph.vertices

Find vertices given edge

v = G.**vertices**(**E**) return the id of the nodes that define edge **E**.

PRM

Probabilistic roadmap navigation class

A concrete subclass of Navigation that implements the probabilistic roadmap navigation algorithm. This performs goal independent planning of roadmaps, and at the query stage finds paths between specific start and goal points.

Methods

plan	Compute the roadmap
path	Compute a path to the goal
visualize	Display the obstacle map
display	Print the parameters in human readable form
char	Convert the parameters to a human readable string

Example

```
load map1
prm = PRM(map);
prm.plan()
prm.path(start, goal)
```

See also

Navigation, DXform, Dstar, PGraph

PRM.PRM

Create a PRM navigation object constructor

p = **PRM**(**map**, **options**) is a probabilistic roadmap navigation object, and **map** is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

'npoints', n	Number of sample points (default 100)
'distthresh', d	Distance threshold, edges only connect vertices closer than d (default 0.3 max(size(occgrid)))

See also

Navigation.Navigation

PRM.char

Convert navigation object to string

P.**char**() is a string representing the state of the navigation object in human-readable form.

See also

PRM.display

PRM.path

Find a path between two points

P.**path**(**start**, **goal**) finds and displays a **path** from **start** to **goal** which is overlaid on the occupancy grid.

x = P.**PATH**(**start**, **goal**) is the path from **start** to **goal** as a $2 \times N$ matrix with columns representing points along the path.

PRM.plan

Create a probabilistic roadmap

P.**plan**() creates the probabilistic roadmap by randomly sampling the free space in the map and building a graph with edges connecting close points. The resulting graph is kept within the object.

PRM.visualize

e

P.**visualize**() displays the occupancy grid with an optional distance field

Options:

'goal'	Superimpose the goal position if set
'nooverlay'	Don't overlay the PRM graph

ParticleFilter

Particle filter class

Monte-carlo based localisation for estimating vehicle position based on odometry and observations of known landmarks.

Methods

run	run the particle filter
plot_xy	display estimated vehicle path
plot_pdf	display particle distribution

Properties

robot	reference to the robot object
sensor	reference to the sensor object
history	vector of structs that hold the detailed information from each time step
nparticles	number of particles used
x	particle states; nparticles x 3
weight	particle weights; nparticles x 1
x_est	mean of the particle population
std	standard deviation of the particle population
Q	covariance of noise added to state at each step
L	covariance of likelihood model
dim	maximum xy dimension

Example

Create a landmark map

```
map = Map(20);
```

and a vehicle with odometry covariance and a driver

```
W = diag([0.1, 1*pi/180].^2);
veh = Vehicle(W);
veh.add_driver( RandomPath(10) );
```

and create a range bearing sensor

```
R = diag([0.005, 0.5*pi/180].^2);
sensor = RangeBearingSensor(veh, map, R);
```


For the particle filter we need to define two covariance matrices. The first is is the covariance of the random noise added to the particle states at each iteration to represent uncertainty in configuration.

```
Q = diag([0.1, 0.1, 1*pi/180]).^2;
```

and the covariance of the likelihood function applied to innovation

```
L = diag([0.1 0.1]);
```

Now construct the particle filter

```
pf = ParticleFilter(veh, sensor, Q, L, 1000);
```

which is configured with 1000 particles. The particles are initially uniformly distributed over the 3-dimensional configuration space.

We run the simulation for 1000 time steps

```
pf.run(1000);
```

then plot the map and the true vehicle path

```
map.plot();
veh.plot_xy('b');
```

and overlay the mean of the particle cloud

```
pf.plot_xy('r');
```

We can plot the standard deviation against time

```
plot(pf.std(1:100,:))
```

The particles are a sampled approximation to the PDF and we can display this as

```
pf.plot_pdf()
```

Acknowledgement

Based on code by Paul Newman, Oxford University, http://www.robots.ox.ac.uk/ pnewman

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

Vehicle, RandomPath, RangeBearingSensor, Map, EKF

ParticleFilter.ParticleFilter

Particle filter constructor

pf = ParticleFilter(**vehicle**, **sensor**, **q**, **L**, **np**) is a particle filter that estimates the state of the **vehicle** with a sensor **sensor**. **q** is covariance of the noise added to the particles at each step (diffusion), **L** is the covariance used in the sensor likelihood model, and **np** is the number of particles.

Notes

- ParticleFilter subclasses Handle, so it is a reference object.
- the initial particle distribution is uniform over the map, essentially the kidnapped robot problem which is unrealistic

See also

Vehicle, Sensor, RangeBearingSensor, Map

ParticleFilter.plot_pdf

Plot particles as a PDF

PF.plot_pdf() plots a sparse PDF as a series of vertical line segments of height equal to particle weight.

ParticleFilter.plot_xy

Plot vehicle position

PF.plot_xy() plot the estimated vehicle path in the xy-plane.

PF.plot_xy(**ls**) as above but the optional line style arguments **ls** are passed to plot.

ParticleFilter.run

Run the particle filter

PF.**run**(**n**) **run** the filter for **n** time steps.

Notes

- all previously estimated states and estimation history is cleared.

Polygon

- General polygon class

p = Polygon(**vertices**);

Methods

plot	Plot polygon
area	Area of polygon
moments	Moments of polygon
centroid	Centroid of polygon
perimeter	Perimter of polygon
transform	Transform polygon
inside	Test if points are inside polygon
intersection	Intersection of two polygons
difference	Difference of two polygons
union	Union of two polygons
xor	Exclusive or of two polygons
display	print the polygon in human readable form
char	convert the polgyon to human readable string

Properties

vertices	List of polygon **vertices**, one per column
extent	Bounding box [minx maxx; miny maxy]
n	Number of **vertices**

Notes

- this is reference class object
- Polygon objects can be used in vectors and arrays

Acknowledgement

The methods inside, intersection, difference, union, and xor are based on code written by:

Kirill K. Pankratov, kirill@plume.mit.edu, http://puddle.mit.edu/ glenn/kirill/saga.html

and require a licence. However the author does not respond to email regarding the licence, so use with care.

Polygon.Polygon

Polygon class constructor

p = **Polygon**(**v**) is a polygon with vertices given by **v**, one column per vertex.

p = **Polygon**(**C**, **wh**) is a rectangle centred at **C** with dimensions **wh**=[WIDTH, HEIGHT].

Polygon.area

Area of polygon

a = P.**area**() is the **area** of the polygon.

Polygon.centroid

Centroid of polygon

x = P.**centroid**() is the **centroid** of the polygon.

Polygon.char

String representation

s = P.**char**() is a compact representation of the polgyon in human readable form.

Polygon.difference

Difference of polygons

d = P.**difference**(**q**) is polygon P minus polygon **q**.

Notes

- If polygons P and **q** are not intersecting, returns coordinates of P.
- If the result **d** is not simply connected or consists of several polygons, resulting vertex list will contain NaNs.

Polygon.display

Display polygon

P.**display**() displays the polygon in a compact human readable form.

See also

Polygon.char

Polygon.inside

Test if points are inside polygon

in = **p**.**inside**(**p**) tests if points given by columns of **p** are **inside** the polygon. The corresponding elements of **in** are either true or false.

Polygon.intersect

Intersection of polygon with list of polygons

i = P.**intersect**(**plist**) indicates whether or not the **Polygon** P intersects with

i(**j**) = 1 if p intersects polylist(**j**), else 0.

Polygon.intersect_line

Intersection of polygon and line segment

i = P.intersect_line(**L**) is the intersection points of a polygon P with the line segment **L**=[x1 x2; y1 y2]. **i** is an $N \times 2$ matrix with one column per intersection, each column is [x y]'.

Polygon.intersection

Intersection of polygons

i = P.**intersection**(**q**) is a **Polygon** representing the **intersection** of polygons P and **q**.

Notes

- If these polygons are not intersecting, returns empty polygon.
- If **intersection** consist of several disjoint polygons (for non-convex P or **q**) then vertices of **i** is the concatenation of the vertices of these polygons.

Polygon.moments

Moments of polygon

a = P.**moments**(**p**, **q**) is the pq'th moment of the polygon.

See also

mpq_poly

Polygon.perimeter

Perimeter of polygon

L = P.**perimeter**() is the **perimeter** of the polygon.

Polygon.plot

Plot polygon

P.**plot**() **plot** the polygon.

P.**plot**(**ls**) as above but pass the arguments **ls** to **plot**.

Polygon.transform

Transformation of polygon vertices

p2 = P.**transform**(**T**) is a new **Polygon** object whose vertices have been transfored by the 3×3 homgoeneous transformation **T**.

Polygon.union

Union of polygons

i = P.**union**(**q**) is a **Polygon** representing the **union** of polygons P and **q**.

Notes

- If these polygons are not intersecting, returns a polygon with vertices of both polygons separated by NaNs.
- If the result P is not simply connected (such as a polygon with a "hole") the resulting contour consist of counter- clockwise "outer boundary" and one or more clock-wise "inner boundaries" around "holes".

Polygon.xor

Exclusive or of polygons

i = P.**union**(**q**) is a **Polygon** representing the **union** of polygons P and **q**.

Notes

- If these polygons are not intersecting, returns a polygon with vertices of both polygons separated by NaNs.
- If the result P is not simply connected (such as a polygon with a "hole") the resulting contour consist of counter- clockwise "outer boundary" and one or more clock-wise "inner boundaries" around "holes".

Quaternion

Quaternion class

A quaternion is a compact method of representing a 3D rotation that has computational advantages including speed and numerical robustness. A quaternion has 2 parts, a scalar s, and a vector v and is typically written: q = s <vx, vy, vz>.

A unit quaternion is one for which $s^2+vx^2+vy^2+vz^2 = 1$. It can be considered as a rotation about a vector in space where q = cos (theta/2) < v sin(theta/2)> where v is a unit vector.

q = **Quaternion**(**x**) is a unit quaternion equivalent to **x** which can be any of:

- orthonormal rotation matrix.
- homogeneous transformation matrix (rotation part only).
- rotation angle and vector

Methods

inv	return inverse of quaterion
norm	return norm of quaternion
unit	return unit quaternion
unitize	unitize this quaternion
plot	same options as trplot()
interp	interpolation (slerp) between q and q2, 0<=s<=1
scale	interpolation (slerp) between identity and q, 0<=s<=1
dot	derivative of quaternion with angular velocity w

Arithmetic operators are overloaded

q+q2	return elementwise sum of quaternions
q-q2	return elementwise difference of quaternions
q*q2	return quaternion product
q*v	rotate vector by quaternion, v is 3×1
q/q2	return q*q2.inv
q^n	return q to power n (integer only)

Properties (read only)

s	real part
v	vector part
R	3×3 rotation matrix
T	4×4 homogeneous transform matrix

Notes

- Quaternion objects can be used in vectors and arrays

See also

trinterp, trplot

Quaternion.Quaternion

Constructor for quaternion objects

q = **Quaternion**() is the identitity quaternion 1<0,0,0> representing a null rotation.

q = **Quaternion**(**q1**) is a copy of the quaternion **q1**

q = **Quaternion**([S V1 V2 V3]) is a quaternion formed by specifying directly its 4 elements

q = **Quaternion**(**s**) is a quaternion formed from the scalar **s** and zero vector part: **s**<0,0,0>

q = **Quaternion**(**v**) is a pure quaternion with the specified vector part: 0<**v**>

q = **Quaternion**(**th**, **v**) is a unit quaternion corresponding to rotation of **th** about the vector **v**.

q = **Quaternion**(**R**) is a unit quaternion corresponding to the orthonormal rotation matrix **R**.

q = **Quaternion**(**T**) is a unit quaternion equivalent to the rotational part of the homogeneous transform **T**.

Quaternion.char

Create string representation of quaternion object

s = Q.**char**() is a compact string representation of the quaternion's value as a 4-tuple.

Quaternion.display

Display the value of a quaternion object

Q.**display**() displays a compact string representation of the quaternion's value as a 4-tuple.

Notes

- this method is invoked implicitly at the command line when the result of an expression is a Quaternion object and the command has no trailing semicolon.

See also

Quaternion.char

Quaternion.double

Convert a quaternion object to a 4-element vector

v = Q.**double**() is a 4-vector comprising the quaternion elements [s vx vy vz].

Quaternion.get.R

Return equivalent orthonormal rotation matrix

R = Q.R is the equivalent 3×3 orthonormal rotation matrix.

Quaternion.get.T

Return equivalent homogeneous transformationmatrix

T = Q.T is the equivalent 4×4 homogeneous transformation matrix.

Quaternion.interp

Interpolate rotations expressed by quaternion objects

qi = Q1.**interp**(**q2**, **R**) is a unit-quaternion that interpolates between Q1 for **R**=0 to **q2** for **R**=1. This is a spherical linear interpolation (slerp) that can be interpretted as interpolation along a great circle arc on a sphere.

If **R** is a vector **qi** is a vector of quaternions, each element corresponding to sequential elements of **R**.

Notes:

- the value of r is clipped to the interval 0 to 1

See also

ctraj, Quaternion.scale

Quaternion.inv

Invert a unit-quaternion

qi = Q.**inv**() is a quaternion object representing the inverse of Q.

Quaternion.minus

Subtract two quaternion objects

Q1-Q2 is the element-wise difference of quaternion elements.

Quaternion.mpower

Raise quaternion to integer power

Q^N is quaternion Q raised to the integer power N, and computed by repeated multiplication.

Quaternion.mrdivide

Compute quaternion quotient.

Q1/Q2	is a quaternion formed by Hamilton product of Q1 and **inv**(Q2)
Q/S	is the element-wise division of quaternion elements by by the scalar S

Quaternion.mtimes

Multiply a quaternion object

Q1*Q2	is a quaternion formed by Hamilton product of two quaternions.
Q*V	is the vector V rotated by the quaternion Q
Q*S	is the element-wise multiplication of quaternion elements by by the scalar S

Quaternion.norm

Compute the norm of a quaternion

qn = q.norm(q) is the scalar **norm** or magnitude of the quaternion **q**.

Quaternion.plot

Plot a quaternion object

Q.**plot**(**options**) plots the quaternion as a rotated coordinate frame.

See also

trplot

Quaternion.plus

Add two quaternion objects

Q1+Q2 is the element-wise sum of quaternion elements.

Quaternion.scale

Interpolate rotations expressed by quaternion objects

qi = Q.**scale**(**R**) is a unit-quaternion that interpolates between identity for **R**=0 to Q for **R**=1. This is a spherical linear interpolation (slerp) that can be interpretted as interpolation along a great circle arc on a sphere.

If **R** is a vector **qi** is a cell array of quaternions, each element corresponding to sequential elements of **R**.

See also

ctraj, Quaternion.interp

Quaternion.unit

Unitize a quaternion

qu = Q.**unit**() is a quaternion which is a unitized version of Q

RRT

Class for rapidly-exploring random tree navigation

A concrete class that implements the **RRT** navigation algorithm. This class subclasses the Navigation class.

Usage for subclass:

rrt = **RRT**(**occgrid**, **options**) create an instance object

rrt	show summary statistics about the object
rrt.visualize()	display the occupancy grid
rrt.plan(goal)	plan a path to coordinate goal
rrt.path(start)	display a path from start to goal
p = **rrt**.path(start)	return a path from start to goal

Options

'npoints', N	Number of nodes in the tree
'time', T	Period to simulate dynamic model toward random point
'xrange', X	Workspace span in x-direction [xmin xmax]
'yrange', Y	Workspace span in y-direction [ymin ymax]
'goal', P	Goal position (1×2) or pose (1×3) in workspace

Notes

- The bicycle model is hardwired into the class (should be a parameter)
- Default workspace is between -5 and +5 in the x- and y-directions

---

RRT.visualize

;

RandomPath

Vehicle driver class

d = **RandomPath**(**dim**, **speed**) returns a "driver" object capable of driving a Vehicle object through random waypoints at constant specified **speed**. The waypoints are positioned inside a region bounded by +/- **dim** in the x- and y-directions.

The driver object is attached to a Vehicle object by the latter's add_driver() method.

Methods

init	reset the random number generator
demand	return speed and steer angle to next waypoint
display	display the state and parameters in human readable form
char	convert the state and parameters to human readable form

Properties

goal	current goal coordinate
veh	the Vehicle object being controlled
dim	dimensions of the work space
speed	speed of travel
closeenough	proximity to waypoint at which next is chosen
randstream	random number stream used for coordinates

Example

```
veh = Vehicle(V);
veh.add_driver( RandomPath(20, 2) );
```


Notes

- it is possible in some cases for the vehicle to move outside the desired region, for instance if moving to a waypoint near the edge, the limited turning circle may cause it to move outside.
- the vehicle chooses a new waypoint when it is closer than property closeenough to the current waypoint.
- uses its own random number stream so as to not influence the performance of other randomized algorithms such as path planning.

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

Vehicle

RandomPath.RandomPath

Create a driver object

d = **RandomPath**(**dim**, **speed**) returns a "driver" object capable of driving a Vehicle object through random waypoints at specified **speed**. The waypoints are positioned inside a region bounded by +/- **dim** in the x- and y-directions.

See also

Vehicle

RandomPath.char

Convert driver parameters and state to a string

s = R.**char**() is a string showing driver parameters and state in in a compact human readable format.

RandomPath.demand

Compute speed and heading to waypoint

[**speed**,**steer**] = R.**demand**() returns the speed and steer angle to drive the vehicle toward the next waypoint. When the vehicle is within R.closeenough a new waypoint is chosen.

See also

Vehicle

RandomPath.display

Display driver parameters and state

R.**display**() **display** driver parameters and state in compact human readable form.

See also

RandomPath.char

RandomPath.init

Reset random number generator

R.**INIT**() resets the random number generator used to create the waypoints. This enables the sequence of random waypoints to be repeated.

See also

randstream

RangeBearingSensor

Range and bearing sensor class

A concrete subclass of Sensor that implements a range and bearing angle sensor that provides robot-centric measurements of the world. To enable this it has references to a map of the world (Map object) and a robot moving through the world (Vehicle object).

Methods

reading	return a random range/bearing observation
h	return the observation for vehicle state xv and feature xf
Hx	return a Jacobian matrix dh/dxv
Hxf	return a Jacobian matrix dh/dxf
Hw	return a Jacobian matrix dh/dw
g	return feature positin given vehicle pose and observation
Gx	return a Jacobian matrix dg/dxv
Gz	return a Jacobian matrix dg/dz

Properties (read/write)

R	measurement covariance matrix
interval	valid measurements returned every interval'th call to reading()

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

Sensor, Vehicle, Map, EKF

RangeBearingSensor.Gx

Jacobian dg/dx

J = S.Gx(**xv**, **z**) returns the Jacobian dg/dxv at the vehicle state **xv**, for measurement **z**. **J** is 2×3.

See also

RangeBearingSensor.g

RangeBearingSensor.Gz

Jacobian dg/dz

J = S.**Gz**(**xv**, **z**) returns the Jacobian dg/dz at the vehicle state **xv**, for measurement **z**. **J** is 2×2.

See also

RangeBearingSensor.g

RangeBearingSensor.Hw

Jacobian dh/dv

J = S.**Hw**(**xv**, **J**) returns the Jacobian dh/dv at the vehicle state **xv**, for map feature **J**. **J** is 2×2.

See also

RangeBearingSensor.h

RangeBearingSensor.Hx

Jacobian dh/dxv

J = S.**Hx**(**xv**, **J**) returns the Jacobian dh/dxv at the vehicle state **xv**, for map feature **J**. **J** is 2×3.

J = S.**Hx**(**xv**, **xf**) as above but for a feature at coordinate **xf**.

See also

RangeBearingSensor.h

RangeBearingSensor.Hxf

Jacobian dh/dxf

J = S.**Hxf**(**xv**, **J**) returns the Jacobian dh/dxv at the vehicle state **xv**, for map feature **J**. **J** is 2×2.

J = S.**Hxf**(**xv**, **xf**) as above but for a feature at coordinate **xf**.

See also

RangeBearingSensor.h

RangeBearingSensor.RangeBearingSensor

Range and bearing sensor constructor

s = **RangeBearingSensor**(**vehicle**, **map**, **R**, **options**) is a range and bearing angle sensor mounted on the Vehicle object **vehicle** and observing the landmark map **map**. The sensor covariance is **R** (2×2) representing range and bearing covariance.

Options

'range', xmax	maximum range of sensor
'range', [xmin xmax]	minimum and maximum range of sensor
'angle', TH	detection for angles betwen -TH to +TH
'angle', [THMIN THMAX]	detection for angles betwen THMIN and THMAX
'skip', I	return a valid reading on every I'th call
'fail', [TMIN TMAX]	sensor simulates failure between timesteps TMIN and TMAX

See also

Sensor, Vehicle, Map, EKF

RangeBearingSensor.g

Compute landmark location

p = S.**g**(**xv**, **z**) is the world coordinate of feature given observation **z** and vehicle state **xv**.

See also

RangeBearingSensor.Gx, RangeBearingSensor.Gz

RangeBearingSensor.h

Landmark range and bearing

z = S.**h**(**xv**, **J**) is range and bearing from vehicle at **xv** to map feature **J**. **z** = [R,theta]

z = S.**h**(**xv**, **xf**) as above but compute range and bearing to a feature at coordinate **xf**.

See also

RangeBearingSensor.Hx, RangeBearingSensor.Hw, RangeBearingSensor.Hxf

---

RangeBearingSensor.reading

Landmark range and bearing

S.**reading**() is a range/bearing observation [Z,J] where Z=[R,THETA] is range and bearing with additive Gaussian noise of covariance R. J is the index of the map feature that was observed. If no valid measurement, ie. no features within range, interval subsampling enabled or simulated failure the return is Z=[] and J=NaN.

See also

RangeBearingSensor.h

Sensor

Sensor superclass

An abstact superclass to represent robot navigation sensors.

s = **Sensor**(**vehicle**, **map**, **R**) is an instance of the **Sensor** object that references the **vehicle** on which the sensor is mounted, the **map** of landmarks that it is observing, and the sensor covariance matrix **R**.

Methods

display	print the parameters in human readable form
char	convert the parameters to a human readable string

Properties

robot	The Vehicle object on which the sensor is mounted
map	The Map object representing the landmarks around the robot

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

EKF, Vehicle, Map

Sensor.Sensor

Sensor object constructor

s = **Sensor**(**vehicle**, **map**) is a sensor mounted on the Vehicle object **vehicle** and observing the landmark map **map**.

Sensor.char

Convert sensor parameters to a string

s = S.**char**() is a string showing sensor parameters in a compact human readable format.

Sensor.display

Display status of sensor object

S.**display**() **display** the state of the sensor object in human-readable form.

Notes

- this method is invoked implicitly at the command line when the result of an expression is a Sensor object and the command has no trailing semicolon.

See also

Sensor.char

Vehicle

Car-like vehicle class

This class models the kinematics of a car-like vehicle (bicycle model). For given steering and velocity inputs it updates the true vehicle state and returns noise-corrupted odometry readings.

veh = **Vehicle**(**v**) creates a **Vehicle** object with odometry covariance **v**, where **v** is a 2×2 matrix corresponding to the odometry vector [dx dtheta].

Methods

init	initialize vehicle state
f	predict next state based on odometry
step	move one time step and return noisy odometry
control	generate the control inputs for the vehicle
update	update the vehicle state
run	run for multiple time steps
Fx	Jacobian of f wrt x
Fv	Jacobian of f wrt odometry noise
gstep	like step() but displays vehicle
plot	plot/animate vehicle on current figure
plot_xy	plot the true path of the vehicle
add_driver	attach a driver object to this vehicle
display	display state/parameters in human readable form
char	convert state/parameters to human readable string

Properties (read/write)

x	true vehicle state 3×1
v	odometry covariance
odometry	distance moved in the last interval
dim	dimension of the robot's world
robotdim	dimension of the robot (for drawing)
L	length of the vehicle (wheelbase)
alphalim	steering wheel limit
maxspeed	maximum vehicle speed
T	sample interval
verbose	verbosity
x_hist	history of true vehicle state $N \times 3$
driver	reference to the driver object
x0	initial state, init() sets x := x0

Examples

Create a vehicle with odometry covariance

```
v = Vehicle( diag([0.1 0.01].^2 );
```

and display its initial state

```
v
```

now apply a speed (0.2m/s) and steer angle (0.1rad) for 1 time step

```
odo = v.update([0.2, 0.1])
```

where odo is the noisy odometry estimate, and the new true vehicle state

```
v
```

We can add a driver object

```
v.add_driver( RandomPath(10) )
```

which will move the vehicle within the region -10<x<10, -10<y<10 which we can see by

```
v.run(1000)
```

which will show an animation of the vehicle moving between randomly selected waypoints.

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

RandomPath, EKF

Vehicle.Fv

Jacobian df/dv

J = V.**Fv**(**x**, **odo**) returns the Jacobian df/dv at the state **x**, for odometry input **odo**. **J** is 3×2.

See also

Vehicle.F, Vehicle.Fx

Vehicle.Fx

Jacobian df/dx

J = V.**Fx**(**x**, **odo**) returns the Jacobian df/dx at the state **x**, for odometry input **odo**. **J** is 3×3.

See also

Vehicle.F, Vehicle.Fv

Vehicle.Vehicle

Vehicle object constructor

v = **Vehicle**(**vact**) creates a **Vehicle** object with actual odometry covariance **vact**, where **vact** is a 2×2 matrix corresponding to the odometry vector [dx dtheta].

Default parameters are:

alphalim	0.5
maxspeed	5
L	1
robotdim	0.2
x0	(0,0,0)

and can be overridden by assigning properties after the object has been created.

Vehicle.add_driver

Add a driver for the vehicle

V.add_driver(**d**) adds a driver object **d** for the vehicle. The driver object has one public method:

[**speed**, **steer**] = **d**.**demand**();

that returns a **speed** and **steer** angle.

See also

RandomPath

Vehicle.char

Convert vehicle parameters and state to a string

s = V.**char**() is a string showing vehicle parameters and state in in a compact human readable format.

Vehicle.control

Compute the control input to vehicle

u = V.**control**(**speed**, **steer**) returns a **control** input (speed,steer) based on provided controls **speed**,**steer** to which speed and steering angle limits have been applied.

u = V.**control**() returns a **control** input (speed,steer) from a "driver" if one is attached, the driver's DEMAND() method is invoked. If no driver is attached then speed and steer angle are assumed to be zero.

See also

RandomPath

Vehicle.display

Display vehicle parameters and state

V.**display**() **display** vehicle parameters and state in compact human readable form.

See also

Vehicle.char

Vehicle.f

Predict next state based on odometry

xn = V.f(**x**, **odo**) predict next state **xn** based on current state **x** and odometry **odo**. **x** is 3×1, **odo** is [distance,change_heading].

xn = V.f(**x**, **odo**, **w**) predict next state **xn** based on current state **x**, odometry **odo**, and odometry noise **w**.

Vehicle.init

Reset state of vehicle object

V.init() sets the state V.x := V.x0

Vehicle.plot

Plot vehicle

V.plot() plots the vehicle on the current axes at a pose given by the current state. If the vehicle has been previously plotted its pose is updated. The vehicle is depicted as a narrow triangle that travels "point first" and has a length V.robotdim.

V.plot(**x**) plots the vehicle on the current axes at the pose **x**.

Vehicle.plot_xy

plot true path followed by vehicle

V.plot_xy() plots the true xy-plane path followed by the vehicle.

V.plot_xy(**ls**) as above but the line style arguments **ls** are passed to plot.

Vehicle.run

Run the vehicle simulation

V.**run**(**n**) **run** the vehicle simulation for **n** timesteps.

p = V.**run**(**n**) **run** the vehicle simulation for **n** timesteps and return the state history as an **n** $\times$ 3 matrix.

See also

Vehicle.step

Vehicle.step

Move the vehicle model ahead one time step

odo = V.**step**(**speed**, **steer**) updates the vehicle state for one timestep of motion at specified **speed** and **steer** angle, and returns noisy odometry.

odo = V.**step**() updates the vehicle state for one timestep of motion and returns noisy odometry. If a "driver" is attached then its DEMAND() method is invoked to compute speed and steer angle. If no driver is attached then speed and steer angle are assumed to be zero.

See also

Vehicle.control, Vehicle.update, Vehicle.add_driver

Vehicle.update

Update the vehicle state

odo = V.**update**(**u**) returns noisy odometry readings (covariance V.V) for motion with **u**=[speed,steer].

---

about

Compact display of variable type

about(**x**) displays a compact line that describes the class and dimensions of **x**.

about **x** as above but this is the command rather than functional form

See also

whos

---

angdiff

Difference of two angles

d = **angdiff**(**th1**, **th2**) returns the difference between angles **th1** and **th2** on the circle. The result is in the interval [-pi pi). If **th1** is a column vector, and **th2** a scalar then return a column vector where **th2** is modulo subtracted from the corresponding elements of **th1**.

d = **angdiff**(**th**) returns the equivalent angle to **th** in the interval [-pi pi).

Return the equivalent angle in the interval [-pi pi).

---

angvec2r

Convert angle and vector orientation to a rotation matrix

R = **angvec2r**(**theta**, **v**) returns an orthonormal rotation matrix, **R**, equivalent to a rotation of **theta** about the vector **v**.

See also

eul2r, rpy2r

---

angvec2tr

Convert angle and vector orientation to a homogeneous transform

T = **angvec2tr**(**theta**, **v**) is a homogeneous transform matrix equivalent to a rotation of **theta** about the vector **v**.

Note

- the translational part is zero.

See also

eul2tr, rpy2tr, angvec2r

---

circle

Compute points on a circle

circle(**C**, **R**, **opt**) plot a **circle** centred at **C** with radius **R**.

x = **circle**(**C**, **R**, **opt**) return an $N \times 2$ matrix whose rows define the coordinates [x,y] of points around the circumferance of a **circle** centred at **C** and of radius **R**.

C is normally 2×1 but if 3×1 then the **circle** is embedded in 3D, and **x** is $N \times 3$, but the **circle** is always in the xy-plane with a z-coordinate of **C**(3).

Options

'n', N	Specify the number of points (default 50)

---

colnorm

Column-wise norm of a matrix

cn = **colnorm**(**a**) returns an $M \times 1$ vector of the normals of each column of the matrix **a** which is $N \times M$.

---

ctraj

Cartesian trajectory between two points

tc = **ctraj**(**T0**, **T1**, **n**) is a Cartesian trajectory from pose **T0** to **T1** with **n** points that follow a trapezoidal velocity profile along the path. The Cartesian trajectory is a 4x4xN matrix, with the last subscript being the point index.

tc = **ctraj**(**T0**, **T1**, **s**) as above but the elements of **s** specify the fractional distance along the path, and these values are in the range [0 1]. The Cartesian trajectory is a 4x4xN matrix, with transform T(:,:,i) corresponding to **s**(i).

See also

mstraj, trinterp, Quaternion.interp, transl

---

delta2tr

Convert differential motion to a homogeneous transform

T = **delta2tr**(**d**) is a homogeneous transform representing differential translation and rotation. The delta vector **d**=(dx, dy, dz, dRx, dRy, dRz) represents an infinitessimal motion, and is an approximation to the spatial velocity multiplied by time.

See also

tr2delta

---

diff2

diff3(**v**)

compute 2-point difference for each point in the vector **v**.

---

distancexform

Distance transform of occupancy grid

d = **distancexform**(**world**, **goal**) is the distance transform of the occupancy grid **world** with respect to the specified goal point **goal** = [X,Y]. The elements of the grid are 0 from free space and 1 for occupied.

d = **distancexform**(**world**, **goal**, **metric**) as above but specifies the distance metric as either 'cityblock' or 'Euclidean'

d = **distancexform**(**world**, **goal**, **metric**, **show**) as above but shows an animation of the distance transform being formed, with a delay of **show** seconds between frames.

Notes

- The Machine Vision Toolbox function imorph is required.

See also

imorph, DXform

---

e2h

Euclidean to homogeneous

---

edgelist

Return list of edge pixels for region

E = **edgelist**(**im**, **seed**) return the list of edge pixels of a region in the image **im** starting at edge coordinate **seed** (i,j). The result **E** is a matrix, each row is one edge point coordinate (x,y).

E = **edgelist**(**im**, **seed**, **direction**) returns the list of edge pixels as above, but the direction of edge following is specified. **direction** == 0 (default) means clockwise, non zero is counter-clockwise. Note that direction is with respect to y-axis upward, in matrix coordinate frame, not image frame.

Notes

- **im** is a binary image where 0 is assumed to be background, non-zero is an object.
- **seed** must be a point on the edge of the region.
- The seed point is always the first element of the returned **edgelist**.

See also

ilabel

---

eul2jac

Jacobian from Euler angle rates to angular velocity

J = **eul2jac**(**eul**) is a 3×3 Jacobian matrix that maps Euler angle rates to angular velocity, and **eul**=[PHI, THETA, PSI]. Used in the creation of an analytical Jacobian.

J = **eul2jac**(**phi**, **theta**, **psi**) as above but the Euler angles are passed as separate arguments.

See also

rpy2jac, SERIALlINK.JACOBN

---

eul2r

Convert Euler angles to homogeneous transformation

R = eul2r(**phi**, **theta**, **psi**) returns an orthonornal rotation matrix equivalent to the specified Euler angles. These correspond to rotations about the Z, Y, Z axes respectively. If **phi**, **theta**, **psi** are column vectors then they are assumed to represent a trajectory and **R** is a three dimensional matrix, where the last index corresponds to rows of **phi**, **theta**, **psi**.

R = eul2r(**eul**) as above but the Euler angles are taken from consecutive columns of the passed matrix **eul** = [**phi theta psi**].

Note

- the vectors **phi**, **theta**, **psi** must be of the same length

See also

eul2tr, rpy2tr, tr2eul

---

eul2tr

Convert Euler angles to homogeneous transform

T = eul2tr(**phi**, **theta**, **psi**) returns a homogeneous tranformation equivalent to the specified Euler angles. These correspond to rotations about the Z, Y, Z axes respectively. If **phi**, **theta**, **psi** are column vectors then they are assumed to represent a trajectory and **T** is a three dimensional matrix, where the last index corresponds to rows of **phi**, **theta**, **psi**.

T = eul2tr(**eul**) as above but the Euler angles are taken from consecutive columns of the passed matrix **eul** = [**phi theta psi**].

Note

- the vectors **phi**, **theta**, **psi** must be of the same length
- the translational part is zero.

See also

eul2r, rpy2tr, tr2eul

---

ftrans

Transform a wrench between coordinate frames

wt = **wtrans**(**T**, **w**) is a wrench in the frame **T** corresponding to the world frame wrench **w**.

The wrenches **w** and **wt** are 6-vectors of the form [Fx Fy Fz Mx My Mz].

See also

tr2delta

---

gauss2d

kernel

k = **gauss2d**(**im**, **c**, **sigma**)

Returns a unit volume Gaussian smoothing kernel. The Gaussian has a standard deviation of **sigma**, and the convolution kernel has a half size of w, that is, **k** is (2W+1) x (2W+1).

If w is not specified it defaults to 2***sigma**.

---

h2e

Homogeneous to Euclidean

---

homline

Homogeneous line from two points

L = **homline**(**x1**, **y1**, **x2**, **y2**) returns a 3×1 vectors which describes a line in homogeneous form that contains the two Euclidean points (**x1**,**y1**) and (**x2**,**y2**).

Homogeneous points X (3×1) on the line must satisfy **L**'*X = 0.

See also

plot_homline

---

homtrans

Apply a homogeneous transformation

p2 = **homtrans**(**T**, **p**) applies homogeneous transformation **T** to the points stored columnwise in **p**.

- If **T** is in SE(2) (3×3) and
  - **p** is $2 \times N$ (2D points) they are considered Euclidean (R^2)
  - **p** is $3 \times N$ (2D points) they are considered projective ($\mathbf{p}^2$)
- If **T** is in SE(3) (4×4) and
  - **p** is $3 \times N$ (3D points) they are considered Euclidean (R^3)
  - **p** is $4 \times N$ (3D points) they are considered projective ($\mathbf{p}^3$)

tp = **homtrans**(**T**, **T1**) applies homogeneous transformation **T** to the homogeneous transformation **T1**, that is **tp**=**T***T1. If **T1** is a 3-dimensional transformation then **T** is applied to each plane as defined by the first two

dimensions, ie. if **T** = $N \times N$ and **T**=NxNxP then the result is NxNxP.

See also

e2h, h2e

---

imeshgrid

Domain matrices for image

[**u**,**v**] = **imeshgrid**(**im**) return matrices that describe the domain of image **im** and can be used for the evaluation of functions over the image. The element **u**(v,u) = u and **v**(v,u) = v.

[**u**,**v**] = **imeshgrid**(**w**, **H**) as above but the domain is **w** $\times$ **H**.

[**u**,**v**] = **imeshgrid**(**size**) as above but the domain is described size which is scalar **size** $\times$ **size** or a 2-vector [**w H**].

See also

meshgrid

---

ishomog

Test if argument is a homogeneous transformation

ishomog(**T**) is true (1) if the argument **T** is of dimension 4×4 or 4x4xN, else false (0).

ishomog(**T**, 'valid') as above, but also checks the validity of the rotation matrix.

See also

isrot, isvec

---

isrot

Test if argument is a rotation matrix

isrot(**R**) is true (1) if the argument is of dimension 3×3, else false (0).

isrot(**R**, 'valid') as above, but also checks the validity of the rotation matrix.

See also

ishomog, isvec

---

isvec

Test if argument is a vector

isvec(**v**) is true (1) if the argument **v** is a 3-vector, else false (0).

isvec(**v**, **L**) is true (1) if the argument **v** is a vector of length **L**, either a row- or column-vector. Otherwise false (0).

See also

ishomog, isrot

---

jsingu

Show the linearly dependent joints in a Jacobian matrix

jsingu(**J**) displays the linear dependency of joints in a Jacobian matrix, which occurs at singularity.

See also

SerialLink.jacobn

---

jtraj

Compute a joint space trajectory between two points

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **n**) is a joint space trajectory **q** where the joint coordinates vary from **q0** to **qf**. A quintic (5th order) polynomial is used with default zero boundary

conditions for velocity and acceleration. Time is assumed to vary from 0 to 1 in **n** steps. Joint velocity and acceleration can be optionally returned as **qd** and **qdd** respectively. The trajectory **q**, **qd** and **qdd** are $M \times$ **n** matrices, with one row per time step, and one column per joint.

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **T**) specifies the trajectory in terms of the time vector **T** and the number of points in the trajectory is equal to the length of **T**.

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **n**, **qd0**, **qdf**) as above but also specifies initial and final joint velocity for the trajectory.

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **T**, **qd0**, **qdf**) as above but specifies initial and final joint velocity for the trajectory and a time vector.

See also

ctraj, SerialLink.jtraj

---

lspb

Linear segment with parabolic blend

[**s**,**sd**,**sdd**] = **lspb**(**s0**, **sf**, **n**) is a scalar trajectory that varies smoothly from **s0** to **sf** in **n** steps using a constant velocity segment and parabolic blends (a trapezoidal path). Velocity and acceleration can be optionally returned as **sd** and **sdd**. The trajectory **s**, **sd** and **sdd** are **n**-vectors.

[**s**,**sd**,**sdd**] = **lspb**(**s0**, **sf**, **T**) as above but specifies the trajectory in terms of the length of the time vector **T**.

[**s**,**sd**,**sdd**] = **lspb**(**s0**, **sf**, **n**, **v**) as above but specifies the velocity of the linear segment which is normally computed automatically.

[**s**,**sd**,**sdd**] = **lspb**(**s0**, **sf**, **T**, **v**) as above but specifies the velocity of the linear segment which is normally computed automatically and a time vector.

Notes

- in all cases if no output arguments are specified **s**, **sd**, and **sdd** are plotted against time.
- for some values of **v** no solution is possible and an error is flagged.

See also

tpoly, jtraj

---

maxfilt

maximum filter

MAXFILT(s [,w])

minimum filter a signal with window of width w (default is 5)

SEE ALSO: medfilt, minfilt

pic 6/93

---

mdl_Fanuc10L

Create kinematic model of Fanuc AM120iB/10L robot

mdl_fanuc10L

Script creates the workspace variable R which describes the kinematic characteristics of a Fanuc AM120iB/10L robot using standard DH conventions.

Also define the workspace vector:

q0 mastering position.

See also

SerialLink, mdl_puma560akb, mdl_stanford, mdl_twolink

Author:

Wynand Swart, Mega Robots CC, P/O Box 8412, Pretoria, 0001, South Africa wynand.swart@gmail.com

---

mdl_MotomanHP6

Create kinematic data of a Motoman HP6 manipulator

mdl_motomanHP6

Script creates the workspace variable R which describes the kinematic characteristics of a Motoman HP6 manipulator using standard DH conventions.

Also define the workspace vector:

q0 mastering position.

See also

SerialLink, mdl_puma560akb, mdl_stanford, mdl_twolink

Author:

Wynand Swart, Mega Robots CC, P/O Box 8412, Pretoria, 0001, South Africa

wynand.swart@gmail.com

---

mdl_S4ABB2p8

Create kinematic model of ABB S4 2.8robot

mdl_s4abb2P8

Script creates the workspace variable R which describes the kinematic characteristics of an ABB S4 2.8 robot using standard DH conventions.

Also define the workspace vector:

q0 mastering position.

See also

SerialLink, mdl_puma560akb, mdl_stanford, mdl_twolink

Author:

Wynand Swart, Mega Robots CC, P/O Box 8412, Pretoria, 0001, South Africa wynand.swart@gmail.com

---

mdl_p8

Create model of Puma robot on an XY base

mdl_p8

Script creates the workspace variable p8 which is an 8-axis robot comprising a Puma 560 robot on an XY base. Joints 1 and 2 are the base, joints 3-8 are the robot arm.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY' configuration
qstretch	arm is stretched out in the X direction
qn	arm is at a nominal non-singular configuration

See also

SerialLink, mdl_puma560

---

mdl_puma560

Create model of Puma 560 manipulator

mdl_puma560

Script creates the workspace variable p560 which describes the kinematic and dynamic characteristics of a Unimation Puma 560 manipulator using standard DH conventions. The model includes armature inertia and gear ratios.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY' configuration
qstretch	arm is stretched out in the X direction
qn	arm is at a nominal non-singular configuration

See also

SerialLink, mdl_puma560akb, mdl_stanford, mdl_twolink

---

mdl_puma560akb

Create model of Puma 560 manipulator

mdl_puma560akb

Script creates the workspace variable p560m which describes the kinematic and dynamic characterstics of a Unimation Puma 560 manipulator modified DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY' configuration
qstretch	arm is stretched out in the X direction

References

Armstrong, Khatib and Burdick "The Explicit Dynamic Model and Inertial Parameters of the Puma 560 Arm" 1986

See also

SerialLink, mdl_puma560, mdl_stanford, mdl_twolink

---

mdl_quadcopter

Dynamic parameters for a quadcopter.

```
mdl_quadcopter
```

Script creates the workspace variable quad which describes the dynamic characterstics of a quadcopter.

Properties

This is a structure with the following elements.

J	Flyer rotational inertia matrix 3×3
h	Height of rotors above CoG 1×1
d	Length of flyer arms 1×1
nb	Number of blades per rotor 1×1
r	Rotor radius 1×1
c	Blade chord 1×1
e	Flapping hinge offset 1×1
Mb	Rotor blade mass 1×1
Mc	Estimated hub clamp mass 1×1
ec	Blade root clamp displacement 1×1
Ib	Rotor blade rotational inertia 1×1
Ic	Estimated root clamp inertia 1×1
mb	Static blade moment 1×1
Ir	Total rotor inertia 1×1
Ct	Non-dim. thrust coefficient 1×1
Cq	Non-dim. torque coefficient 1×1
sigma	Rotor solidity ratio 1×1
thetat	Blade tip angle 1×1
theta0	Blade root angle 1×1
theta1	Blade twist angle 1×1
theta75	3/4 blade angle 1×1
thetai	Blade ideal root approximation 1×1
a	Lift slope gradient 1×1
A	Rotor disc area 1×1
gamma	Lock number 1×1

References

- P.Pounds; Design, Construction and Control of a Large Quadrotor micro air vehicle. PhD thesis, Australian National University, 2007. http://www.eng.yale.edu/pep5/P_Pounds_Thesis_2008.pdf

See also

sl_quadcopter

---

mdl_stanford

Create model of Stanford arm

mdl_stanford

Script creates the workspace variable stanf which describes the kinematic and dynamic characteristics of the Stanford (Scheinman) arm.

Also define the vectors:

qz zero joint angle configuration.

Note

- gear ratios not currently known, though reflected armature inertia is known, so gear ratios are set to 1.

References

- Kinematic data from ”Modelling, Trajectory calculation and Servoing of a computer controlled arm”. Stanford AIM-177. Figure 2.3
- Dynamic data from “Robot manipulators: mathematics, programming and control” Paul 1981, Tables 6.4, 6.6

See also

SerialLink, mdl_puma560, mdl_puma560akb, mdl_twolink

---

mdl_twolink

Create model of a simple 2-link mechanism

mdl_twolink

Script creates the workspace variable tl which describes the kinematic and dynamic characteristics of a simple planar 2-link mechanism.

Also define the vector:

qz corresponds to the zero joint angle configuration.

Notes

- It is a planar mechanism operating in the XY (horizontal) plane and is therefore not affected by gravity.
- Assume unit length links with all mass (unity) concentrated at the joints.

References

- Based on Fig 3-6 (p73) of Spong and Vidyasagar (1st edition).

See also

SerialLink, mdl_puma560, mdl_stanford

---

mlabel

for mplot style graph

mlabel(lab1 lab2 lab3)

---

mplot

multiple data

Plot y versus t in multiple windows.

```
MPLOT(y)
MPLOT(y, n)
MPLOT(y, n, {labels})
```

Where y is multicolumn data and first column is time. n is a row vector specifying which variables to plot (1 is first data column, or y(:,2)). labels is a cell array of labels for the subplots.

```
MPLOT(t, y)
MPLOT(t, y, n)
MPLOT(t, y, {labels})
```

Where y is multicolumn data and t is time. n is a row vector specifying which variables to plot (1 is first data column, or y(:,2)). labels is a cell array of labels for the subplots.

```
MPLOT(S)
```

Where S is a structure and one element 't' is assumed to be time. Plot

all other vectors versus time in subplots. Subplots are labelled as per the data fields.

---

mstraj

Multi-segment multi-axis trajectory

traj = mstraj(**segments**, **qdmax**, **q0**, **dt**, **tacc**) is a multi-segment trajectory based on via points **segments** and velocity limits **qdmax**. The path comprises linear segments with polynomial blends. The output trajectory is an $M \times N$ matrix, with one row per time step, and one column per axis.

- **segments** is an $N \times M$ matrix of via points, 1 row per via point, one column per axis. The last via point is the destination.
- **qdmax** is a row N-vector of axis velocity limits, or a column M-vector of segment times
- **q0** is an N-vector of initial axis coordinates
- **dt** is the time step
- **tacc** is the acceleration time. If scalar this acceleration time is applied to all segment transitions, if an N-vector it specifies the acceleration time for each segment. **tacc**(i) is the acceleration time for the transition from segment i to segment i+1. **tacc**(1) is also the acceleration time at the start of segment 1.

traj = mstraj(**segments**, **qdmax**, **q0**, **dt**, **tacc**, **qd0**, **qdf**) as above but additionally specifies the initial and final axis velocities as N-vectors.

Notes

- can be used to create joint space trajectories
- can be used to create Cartesian trajectories with the "axes" assigned to translation and orientation in RPY or Euler angle form.:w

See also

mstraj, lspb, ctraj

---

mtools

simple/useful tools to all windows in figure

---

mtraj

Multi-axis trajectory between two points

[**q**,**qd**,**qdd**] = **mtraj**(**tfunc**, **q0**, **qf**, **n**) is multi-axis trajectory **q** varying from state **q0** to **qf** according to the scalar trajectory function **tfunc** in **n** steps. Joint velocity and acceleration can be optionally returned as **qd** and **qdd** respectively. The trajectory **q**, **qd** and **qdd** are $M \times$ **n** matrices, with one row per time step, and one column per axis.

The shape of the trajectory is given by the scalar trajectory function **tfunc**

[**s**,**sd**,**sdd**] = **tfunc**(**s0**, **sf**, **n**);

and possible values of **tfunc** include @lspb for a trapezoidal trajectory, or @tpoly for a polynomial trajectory.

[**q**,**qd**,**qdd**] = **mtraj**(**tfunc**, **q0**, **qf**, **T**) as above but specifies the trajectory length in terms of the length of the time vector **T**.

Notes

- when **tfunc** is @tpoly the result is similar to JTRAJ except that no initial velocities can be specified. JTRAJ is computationally a little more efficient.

See also

jtraj, mstraj, lspb, tpoly

---

norm2

columnwise norm

n = **norm2**(**m**) **n** = **norm2**(a, b)

---

numcols

Return number of columns in matrix

nc = **numcols**(**m**) returns the number of columns in the matrix **m**.

See also

numrows

---

numrows

Return number of rows in matrix

nr = **numrows**(**m**) returns the number of rows in the matrix **m**.

See also

numcols

---

oa2r

Convert orientation and approach vectors to rotation matrix

R = **oa2r**(**o**, **a**) is a rotation matrix for the specified orientation and approach vectors formed from 3 vectors such that **R** = [N **o** **a**] and N = **o** x **a**.

Notes

- The submatrix is guaranteed to be orthonormal so long as **o** and **a** are not parallel.

See also

rpy2r, eul2r, oa2tr

---

oa2tr

Convert orientation and approach vectors to homogeneous transformation

T = **oa2tr**(**o**, **a**) is a homogeneous tranformation for the specified orientation and approach vectors. The rotation submatrix is formed from 3 vectors such that R = [N,**o**,**a**] and N = **o** x **a**.

Notes

- The submatrix is guaranteed to be orthonormal so long as **o** and **a** are not parallel.
- the translational part is zero.

See also

rpy2tr, eul2tr, oa2r

---

plot2

Plot trajectories

plot2(**p**) plots a line with coordinates taken from successive rows of **p**. **p** can be $N \times 2$ or $N \times 3$.

If **p** has three dimensions, ie. Nx2xM or Nx3xM then the M trajectories are overlaid in the one plot.

plot2(**p**, **ls**) as above but the line style arguments **ls** are passed to plot.

See also

plot

---

plot_box

a box on the current plot

PLOT_BOX(**b**, **ls**) draws a box defined by **b**=[XL XR; YL YR] with optional Matlab linestyle options **ls**.

PLOT_BOX(**x1**,**y1**, **x2**,**y2**, **ls**) draws a box with corners at (**x1**,**y1**) and (**x2**,**y2**), and optional Matlab linestyle options **ls**.

PLOT_BOX('centre', P, 'size', W, **ls**) draws a box with center at P=[X,Y] and with dimensions W=[WIDTH HEIGHT].

PLOT_BOX('topleft', P, 'size', W, **ls**) draws a box with top-left at P=[X,Y] and with dimensions W=[WIDTH HEIGHT].

---

plot_circle

Draw a circle on the current plot

PLOT_CIRCLE(**C**, **R**, **options**) draws a circle on the current plot with centre **C**=[X Y] and radius **R**. If **C**=[X Y Z] the circle is drawn in the XY-plane at height Z.

If **C** ($2 \times N$ or $3 \times N$) and **R** ($1 \times N$) then a set of N circles are drawn with centre and radius taken from the columns of **C** and **R**.

Options

'edgecolor'	the color of the circle's edge, Matlab color spec
'fillcolor'	the color of the circle's interior, Matlab color spec
'alpha'	transparency of the filled circle: 0=transparent, 1=solid.

Notes

- the option can be either a simple linespec (eg. 'r', 'g:') for a non-filled circle, or a set of name, value pairs that are passed to plot.

Examples

```
plot_circle(c, r, 'r');
plot_circle(c, r, 'fillcolor', 'b');
plot_circle(c, r, 'edgecolor', 'g', 'LineWidth', 5);
```


See also

plot

---

plot_ellipse

Draw an ellipse on the current plot

PLOT_ELLIPSE(**a**, **ls**) draws an ellipse defined by X'AX = 0 on the current plot, centred at the origin, with Matlab line style **ls**.

PLOT_ELLIPSE(**a**, **C**, **ls**) as above but centred at **C**=[X,Y]. current plot. If **C**=[X,Y,Z] the ellipse is parallel to the XY plane but at height Z.

---

plot_ellipse_inv

Plot an ellipse

plot_ellipse(**a**, **xc**, **ls**)

ls is the standard line styles.

---

plot_frame

Plot a coordinate frame represented by a homogeneous transformation

trplot(**T**, **options**) draws a coordinate frame corresponding to the homogeneous transformation **T**.

Options

'color', c Specify color of the axes, Matlab colorspec
'axes' 'axis'

‘name’, n Specify the name of the coordinate frame,
‘framename’, n ‘text_opts’, to

‘view’, v Specify the view angle for the Matlab axes
‘width’, w ‘arrow’ ’length’, l Specify length of the axes (default 1)

Examples:

```
trplot( T, 'color', 'r');
trplot( T, 'color', 'r', 'name', 'B')
```

plot_homline

Draw a line in homogeneous form

H = PLOT_HOMLINE(**L**, **ls**) draws a line in the current figure **L**.X = 0. The current axis limits are used to determine the endpoints of the line. Matlab line specification **ls** can be set.

The return argument is a vector of graphics handles for the lines.

See also

homline

plot_point

point features

PLOT_POINT(**p**, **options**) adds point markers to a plot, where **p** is $2 \times N$ and each column is the point coordinate.

Options

‘textcolor’, colspec	Specify color of text
‘textsize’, size	Specify size of text
‘bold’	Text in bold font.
‘printf’, fmt, data	Label points according to printf format string and corresponding element of data
‘sequence’	Label points sequentially

Additional options are passed through to PLOT for creating the marker.

See also

plot, text

---

plot_poly

Plot a polygon

plotpoly(**p**, **options**) plot a polygon defined by columns of **p** which can be $2 \times N$ or $3 \times N$.

options

'fill'	the color of the circle's interior, Matlab color spec
'alpha'	transparency of the filled circle: 0=transparent, 1=solid.

See also

plot, patch, Polygon

---

plot_sphere

Plot spheres

PLOT_SPHERE(**C**, **R**, **color**) add spheres to the current figure. **C** is the centre of the sphere and if its a $3 \times N$ matrix then N spheres are drawn with centres as per the columns. **R** is the radius and **color** is a Matlab color spec, either a letter or 3-vector.

H = PLOT_SPHERE(**C**, **R**, **color**) as above but returns the handle(s) for the spheres.

H = PLOT_SPHERE(**C**, **R**, **color**, **alpha**) as above but **alpha** specifies the opacity of the sphere were 0 is transparant and 1 is opaque. The default is 1.

NOTES

- The sphere is always added, irrespective of figure hold state.
- The number of vertices to draw the sphere is hardwired.

plotbotopt

Define default options for robot plotting

Default options for robot/plot function.

See also

SerialLink.plot

plotp

Plot trajectories

plotp(**p**) plots a set of points **p**, which by Toolbox convention are stored one per column. **p** can be $N \times 2$ or $N \times 3$. By default a linestyle of 'bx' is used.

plotp(**p**, **ls**) as above but the line style arguments **ls** are passed to plot.

See also

plot, plot2

qplot

Plot joint angles

qplot(**q**) is a convenience function to plot joint angle trajectories for a 6-axis robot. **q** is $N \times 6$, and the first three joints are shown as solid lines, the last three joints are shown as dashed lines. A legend is also displayed.

qplot(**T**, **q**) displays the joint angle trajectory versus time **T**.

See also

jtraj

---

r2t

Convert rotation matrix to a homogeneous transform

T = **r2t**(**R**) is a homogeneous transform equivalent to an orthonormal rotation matrix **R** with a zero translational component.

Notes

- functions for **T** in SE(2) or SE(3)
  - if **R** is 2×2 then **T** is 3×3, or
  - if **R** is 3×3 then **T** is 4×4.
- translational component is zero

See also

t2r

---

ramp

create a ramp vector

ramp(**n**) output a vector of length **n** that ramps linearly from 0 to 1

ramp(**v**) as above but vector is same length as **v**

ramp(**v**, **d**) as above but elements increment by **d**.

See also

linspace

rotx

Rotation about X axis

R = **rotx**(**theta**) is a rotation matrix representing a rotation of **theta** about the x-axis.

See also

roty, rotz, angvec2r

roty

Rotation about Y axis

R = **roty**(**theta**) is a rotation matrix representing a rotation of **theta** about the y-axis.

See also

rotx, rotz, angvec2r

rotz

Rotation about Z axis

R = **rotz**(**theta**) is a rotation matrix representing a rotation of **theta** about the z-axis.

See also

rotx, roty, angvec2r

rpy2jac

Jacobian from RPY angle rates to angular velocity

J = **rpy2jac**(**rpy**) is a 3×3 Jacobian matrix that maps roll-pitch-yaw angle rates to angular velocity, and **rpy**=[R,P,Y]. Used in the creation of the analytical Jacobian.

J = **rpy2jac**(**R**, **p**, **y**) as above but the roll-pitch-yaw angles are passed as separate arguments.

See also

eul2jac, SERIALLINK.JACOBN

---

rpy2r

Roll-pitch-yaw angles to rotation matrix

R = **rpy2r**(**rpy**) is an orthonormal rotation matrix equivalent to the specified roll, pitch, yaw angles which correspond to rotations about the X, Y, Z axes respectively. If **rpy** has multiple rows they are assumed to represent a trajectory and **R** is a three dimensional matrix, where the last index corresponds to the rows of **rpy**.

R = **rpy2r**(**roll**, **pitch**, **yaw**) as above but the roll-pitch-yaw angles are passed as separate arguments.

If **roll**, **pitch** and **yaw** are column vectors then they are assumed to represent a trajectory and **R** is a three dimensional matrix, where the last index corresponds to the rows of **roll**, **pitch**, **yaw**.

Note

- in previous releases (<8) the angles corresponded to rotations about ZYX.
- many texts (Paul, Spong) use the rotation order ZYX.

See also

tr2rpy, eul2tr

---

rpy2tr

Roll-pitch-yaw angles to homogeneous transform

T = rpy2tr(**rpy**) is an orthonormal rotation matrix equivalent to the specified roll, pitch, yaw angles which correspond to rotations about the X, Y, Z axes respectively. If **rpy** has multiple rows they are assumed to represent a trajectory and R is a three dimensional matrix, where the last index corresponds to the rows of **rpy**.

T = rpy2tr(**roll**, **pitch**, **yaw**) as above but the roll-pitch-yaw angles are passed as separate arguments.

If **roll**, **pitch** and **yaw** are column vectors then they are assumed to represent a trajectory and R is a three dimensional matrix, where the last index corresponds to the rows of **roll**, **pitch**, **yaw**.

Note

- in previous releases (<8) the angles corresponded to rotations about ZYX.
- many texts (Paul, Spong) use the rotation order ZYX.

See also

tr2rpy, eul2tr

---

rt2tr

Convert rotation and translation to homogeneous transform

TR = rt2tr(**R**, **t**) is a homogeneous transformation matrix formed from an orthonormal rotation matrix **R** and a translation vector **t**.

Notes

- functions for **R** in SO(2) or SO(3)
  - If **R** is 2×2 and **t** is 2×1, then **TR** is 3×3
  - If **R** is 3×3 and **t** is 3×1, then **TR** is 4×4
- the validity of **R** is not checked

See also

t2r, r2t, tr2rt

---

rtdemo

Robot toolbox demonstrations

Displays popup menu of toolbox demonstration scripts that illustrate:

- homogeneous transformations
- trajectories
- forward kinematics
- inverse kinematics
- robot animation
- inverse dynamics
- forward dynamics

The scripts require the user to periodically hit <Enter> in order to move through the explanation. Set PAUSE OFF if you want the scripts to run completely automatically.

---

se2

Create planar translation and rotation transformation

T = **se2**(**x**, **y**, **theta**) is a 3×3 homogeneous transformation SE(2) representing translation **x** and **y**, and rotation **theta** in the plane.

T = **se2**(**xy**) as above where **xy**=[**x**,**y**] and rotation is zero

T = **se2**(**xy**, **theta**) as above where **xy**=[**x**,**y**]

T = **se2**(**xyt**) as above where **xyt**=[**x**,**y**,**theta**]

See also

trplot2

skew

Create skew-symmetric matrix

s = **skew**(**v**) is a **skew**-symmetric matrix and **v** is a 3-vector.

See also

vex

t2r

Return rotational submatrix of a homogeneous transformation

R = **t2r**(**T**) is the orthonormal rotation matrix component of homogeneous transformation matrix **T**:

Notes

- functions for **T** in SE(2) or SE(3)
  - If **T** is 4×4, then **R** is 3×3.
  - If **T** is 3×3, then **R** is 2×2.
- the validity of rotational part is not checked

See also

r2t, tr2rt, rt2tr

tb_optparse

Standard option parser for Toolbox functions

[**optout**,**args**] = TB_OPTPARSE(**opt**, **arglist**) is a generalized option parser for Toolbox functions. It supports options that have an assigned value, boolean or enumeration types (string or int).

The software pattern is:

```
function(a, b, c, varargin)
opt.foo = true;
opt.bar = false;
opt.blah = [];
opt.choose = {'this', 'that', 'other'};
opt.select = {'#no', '#yes'};
opt = tb_optparse(opt, varargin);
```

Optional arguments to the function behave as follows:

'foo'	sets opt.foo <- true
'nobar'	sets opt.foo <- false
'blah', 3	sets opt.blah <- 3
'blah', x,y	sets opt.blah <- x,y
'that'	sets opt.choose <- 'that'
'yes'	sets opt.select <- 2 (the second element)

and can be given in any combination.

If neither of 'this', 'that' or 'other' are specified then opt.choose <- 'this'. If neither of 'no' or 'yes' are specified then opt.select <- 1.

Note:

- that the enumerator names must be distinct from the field names.
- that only one value can be assigned to a field, if multiple values

```
are required they must be converted to a cell array.
```

The allowable options are specified by the names of the fields in the structure opt. By default if an option is given that is not a field of opt an error is declared.

Sometimes it is useful to collect the unassigned options and this can be achieved using a second output argument

```
[opt,arglist] = tb_optparse(opt, varargin);
```

which is a cell array of all unassigned arguments in the order given in varargin.

The return structure is automatically populated with fields: verbose and debug. The following options are automatically parsed:

'verbose'	sets opt.verbose <- true
'debug', N	sets opt.debug <- N
'setopt', S	sets opt <- S
'showopt'	displays opt and arglist

tpoly

Generate scalar polynomial trajectory

[**s**,**sd**,**sdd**] = **tpoly**(**s0**, **sf**, **n**) is a trajectory of a scalar that varies smoothly from **s0** to **sf** in **n** steps using a quintic (5th order) polynomial. Velocity and acceleration can be optionally returned as **sd** and **sdd**. The trajectory **s**, **sd** and **sdd** are **n**-vectors.

[**s**,**sd**,**sdd**] = **tpoly**(**s0**, **sf**, **T**) as above but specifies the trajectory in terms of the length of the time vector **T**.

---

tr2angvec

Convert rotation matrix to angle-vector form

[**theta**,**v**] = **tr2angvec**(**R**) is a rotation of **theta** about the vector **v** equivalent to the orthonormal rotation matrix **R**.

[**theta**,**v**] = **tr2angvec**(**T**) is a rotation of **theta** about the vector **v** equivalent to the rotational component of the homogeneous transform **T**.

Notes

- If no output arguments are specified the result is displayed.

See also

angvec2r, angvec2tr

---

tr2delta

Convert homogeneous transform to differential motion

d = **tr2delta**(**T0**, **T1**) is the differential motion corresponding to infinitessimal motion from pose **T0** to **T1** which are homogeneous transformations. **d**=(dx, dy, dz, dRx, dRy, dRz) and is an approximation to the average spatial velocity multiplied by time.

d = **tr2delta**(**T**) is the differential motion corresponding to the infinitessimal relative pose **T** expressed as a homogeneous transformation.

Notes

- **d** is only an approximation to the described by **T**, and assumes that **T0** **T1** or **T** eye(4,4).

See also

delta2tr, skew

---

tr2eul

Convert homogeneous transform to Euler angles

eul = **tr2eul**(**T**, **options**) are the ZYZ Euler angles expressed as a row vector corresponding to the rotational part of a homogeneous transform **T**. The 3 angles **eul**=[PHI,THETA,PSI] correspond to sequential rotations about the Z, Y and Z axes respectively.

eul = **tr2eul**(**R**, **options**) are the ZYZ Euler angles expressed as a row vector corresponding to the orthonormal rotation matrix **R**.

Notes

- There is a singularity for the case where THETA=0 in which case PHI is arbitrarily set to zero and PSI is the sum (PHI+PSI).
- If **R** or **T** represents a trajectory (has 3 dimensions), then each row of **eul** corresponds to a step of the trajectory.

Options:

'deg' Compute angles in degrees (radians default)

See also

eul2tr, tr2rpy

---

tr2jac

Jacobian for differential motion

J = **tr2jac**(**T**) is a 6×6 Jacobian matrix that maps spatial velocity or differential motion from the world frame to the frame represented by the homogeneous transform **T**.

See also

wtrans, tr2delta, delta2tr

---

tr2rpy

Convert a homogeneous transform to roll-pitch-yaw angles

rpy = **tr2rpy**(**T**, **options**) are the roll-pitch-yaw angles expressed as a row vector corresponding to the rotation part of a homogeneous transform **T**. The 3 angles **rpy**=[R,P,Y] correspond to sequential rotations about the X, Y and Z axes respectively.

rpy = **tr2rpy**(**R**, **options**) are the roll-pitch-yaw angles expressed as a row vector corresponding to the orthonormal rotation matrix **R**.

If **R** or **T** represents a trajectory (has 3 dimensions), then each row of **rpy** corresponds to a step of the trajectory.

Options

'deg'	Compute angles in degrees (radians default)
'zyx'	Return solution for sequential rotations about Z, Y, X axes (Paul book)

Notes

- There is a singularity for the case where THETA=0 in which case PHI is arbitrarily set to zero and PSI is the sum (PHI+PSI).
- Note that textbooks (Paul, Spong) use the rotation order ZYX.

See also

rpy2tr, tr2eul

---

tr2rt

Convert homogeneous transform to rotation and translation

[**R**,**t**] = **tr2rt**(**TR**) split a homogeneous transformation matrix into an orthonormal rotation matrix **R** and a translation vector **t**.

Notes

- Functions for **TR** in SE(2) or SE(3)
  - If **TR** is 4×4, then **R** is 3×3 and T is 3×1.
  - If **TR** is 3×3, then **R** is 2×2 and T is 2×1.
- The validity of **R** is not checked.

See also

rt2tr, r2t, t2r

---

tranimate

Animate a coordinate frame

tranimate(**p1**, **p2**, **options**) animates a coordinate frame moving from pose **p1** to pose **p2**. Pose can be represented by:

- homogeneous transformation matrix 4×4
- orthonormal rotation matrix 3×3
- Quaternion

tranimate(**p**, **options**) animates a coordinate frame moving from the identity pose to the pose **p** represented by any of the types listed above.

tranimate(**ps**, **options**) animates a trajectory, where **ps** is any of

- homogeneous transformation matrix sequence 4x4xn
- orthonormal rotation matrix sequence 3x3xn
- quaternion array n

Options

'fps', fps	Number of frames per second to display (default 10)
'nsteps', n	The number of steps along the path (default 50)

See also

trplot

---

transl

Create translational transform

T = **transl**(**x**, **y**, **z**) is a homogeneous transform representing a pure translation.

T = **transl**(**p**) is a homogeneous transform representing a translation or point **p**=[**x**,**y**,**z**]. If **p** is an $M \times 3$ matrix **transl** returns a 4x4xM matrix representing a sequence of homogenous transforms such that **T**(:,:,i) corresponds to the i'th row of **p**.

p = **transl**(**T**) is the translational part of a homogenous transform as a 3-element column vector. If **T** has three dimensions, ie. 4x4xN then **T** is considered a homgoeneous transform sequence and returns an $N \times 3$ matrix where each row is the translational component of the corresponding transform in the sequence.

Notes

- somewhat unusually this function performs a function and its inverse. An historical anomaly.

See also

ctraj

---

trinterp

Interpolate homogeneous transformations

T = **trinterp**(**T0**, **T1**, **s**) is a homogeneous transform interpolation between **T0** when **s**=0 to **T1** when **s**=1. Rotation is interpolated using quaternion spherical linear interpolation. If **s** is an N-vector then **T** is a 4x4xN matrix where the transform **T**(:,:,i) corresponds to **s**(i).

T = **trinterp**(**T**, **s**) is a transform that varies from the identity matrix when **s**=0 to **T** when R=1. If **s** is an N-vector then **T** is a 4x4xN matrix where the transform **T**(:,:,i) corresponds to **s**(i).

See also

ctraj, quaternion

---

trnorm

Normalize a homogeneous transform

tn = **trnorm**(**T**) is a normalized homogeneous transformation matrix in which the rotation submatrix is guaranteed to be a proper orthogonal matrix. The O and A vectors are normalized and the normal vector is formed from O x A.

Notes

- Used to prevent finite word length arithmetic causing transforms to

become 'unnormalized'.

See also

oa2tr

---

trotx

Rotation about X axis

T = **trotx**(**theta**) is a homogeneous transformation representing a rotation of **theta** about the x-axis.

Notes

- translational component is zero

See also

rotx, troty, trotz

---

troty

Rotation about Y axis

T = **troty**(**theta**) is a homogeneous transformation representing a rotation of **theta** about the y-axis.

Notes

- translational component is zero

See also

roty, trotx, trotz

---

trotz

Rotation about Z axis

T = **trotz**(**theta**) is a homogeneous transformation representing a rotation of **theta** about the z-axis.

Notes

- translational component is zero

See also

rotz, trotx, troty

---

trplot

Draw a coordinate frame

trplot(**T**, **options**) draws a 3D coordinate frame represented by the homogeneous transform **T**.

trplot(**R**, **options**) draws a 3D coordinate frame represented by the orthonormal rotation matrix **R**.

Options

'color', c	The color to draw the axes, Matlab colorspec.
'axes'	Set dimensions of the Matlab axes
'frame', f	The name which appears on the axis labels and the frame itself
'text_opts', opt	A cell array of Matlab text properties
'arrow'	Use arrows rather than line segments for the axes
'width', w	Width of arrow tips
'handle', h	Draw in the Matlab axes specified by h

See also

trplot2, tranimate

---

trplot2

Plot a planar transformation

trplot2(**T**, **options**) draws a 2D coordinate frame represented by the homogeneous transform **T**.

Options

'color', c	The color to draw the axes, Matlab colorspec.
'axes'	Set dimensions of the Matlab axes
'frame', f	The name which appears on the axis labels and the frame itself
'text_opts', opt	A cell array of Matlab text properties
'arrow'	Use arrows rather than line segments for the axes
'width', w	Width of arrow tips
'handle', h	Draw in the Matlab axes specified by h

See also

trplot

---

trprint

Compact display of homogeneous transformation

trprint(**T**, **options**) displays the homogoneous transform in a compact single-line format. If **T** is a homogeneous transform sequence then print each element is printed on a separate line.

trprint T is the command line form of above, and displays in RPY format.

Options

'rpy'	display with rotation in roll/pitch/yaw angles (default)
'euler'	display with rotation in ZYX Euler angles
'angvec'	display with rotation in angle/vector format
'radian'	display angle in radians (default is degrees)
'fmt', f	use format string f for all numbers, (default %g)
'label', l	display the text before the transform

See also

tr2eul, tr2rpy, tr2angvec

---

unit

Unitize a vector

vn = **unit**(**v**) is a **unit** vector parallel to **v**.

Note

- fails for the case where norm(**v**) is zero.

---

usefig

a named figure or create a new figure

usefig('Foo')

make figure 'Foo' the current figure, if it doesn't exist create it.

h = usefig('Foo') as above, but returns the figure handle

---

vex

Convert skew-symmetric matrix to vector

v = **vex**(**s**) is the vector which has the skew-symmetric matrix **s**.

Notes

- No checking is done to ensure that the matrix is skew-symmetric.
- The function takes the mean of the two elements that correspond to each unique element of the matrix, ie. vx = 0.5*(**s**(3,2)-**s**(2,3))

See also

skew

---

xaxis

X-axis scaling

xaxis(**max**) **xaxis**([min **max**]) **xaxis**(min, **max**)

xaxis restore automatic scaling for this axis

---

yaxis

Y-axis scaling

yayis(**max**) **yayis**(min, **max**)

YAXIS restore automatic scaling for this axis
